Introducing

Z80

Assembly Language Programming

Ian R. Sinclair

Newnes Microcomputer Books

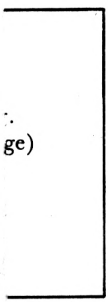
Introducing Z80 Assembly Language Programming

Ian R. Sinclair

Newnes Technical

ɒ, Wellington

gland



ge)

, Kent

Preface

There has never been any shortage of information on any of the major microprocessor chips, but most of the books that have been previously available on the 6502 and the Z-80 have been rather specialised. Specialism has taken two forms, of which one is explaining assembly language with a view to the book being used as a reference work by writers of interpreters and compilers. The other option is a book which is specific to the use of one machine, such as the TRS-80 or the ZX-81. This book is designed as an introduction to assembly language programming of the Z-80, assuming only that the reader has some experience of using a language such as BASIC, and enough background knowledge of microprocessors to understand words such as 'address' and 'data'. Some reference to specific hardware is needed, because a Z-80 on its own is of no use and most writers of assembly language will be writing for their own computer, but every care has been taken to avoid giving details that apply to one machine in such a form that the reader would not realise that these details applied to that one machine.

The Z-80 has now been manufactured for a considerable time (several years; a long time in IC terms), and its use is by now very well established. As a result, no book can possibly quote examples of wholly original subroutines, and I would not claim any originality for any of the examples shown here. Most of them have, however, been taken from my own assembly language programs, written either for the TRS-80 or on MENTA, and the few that are not of my own making and for which I know a source have been acknowledged.

Unlike many authors of books on Z-80 assembly language programming, I have laid very little emphasis on arithmetical routines. These are of little interest to the beginner, and generally concern only the writer of a compiler or interpreter – who is not likely to be reading this book. For that reason also, I have emphasised the other aspects of Z-80 programming which are more likely to be used by the hobby user or the user interested in machine control. I have avoided long examples as far as possible, because it is much more useful for the newcomer to assembly language to design his/her own program than to wade through anyone else's work. Once general methods are known, then practice, using this

book as a guide, is the best path to achievement in assembly language skills.

I am grateful to a large number of people who have contributed to this book, particularly to Phil Chapman of Newnes Technical Books for the initial idea and continual encouragement, and to Barry Savage of Dataman Designs, who very kindly lent me the MENTA assembler unit to keep in touch with Z-80 assembly language when I replaced my TRS-80 with a BBC machine. I must also mention the tolerance of my wife, faced with a room full of buzzing printers, and the cheerful co-operation of countless members of the TRS-80 Users' Group, particularly Laurie Shields.

Ian Sinclair

Contents

1

Why and how

Any system that uses a microprocessor, from a simple controller to a large computer, will require the microprocessor to be controlled by a program, because the action of the microprocessor is completely governed by a program. A microprocessor is programmed by applying sets of electrical signals to it, and each different set of signals can be represented by a number, so that a microprocessor program can be written down as a series of numbers. This set of numbers is called 'machine code' or 'object code', and each number represents a set of signals that will be applied to the microprocessor.

Computers are not generally programmed in this way by the users. Using programs that consist of strings of numbers is not the simplest way of programming, and machine code is referred to as a 'low-level' language. It is, in fact, very close to the most primitive method imaginable, which is to control the action of the microprocessor by the settings of switches. Users and programmers of computers need to be able to ignore the details of what the microprocessor is doing, and concentrate on designing programs to solve specific problems, so that 'high-level' languages such as COBOL, FORTRAN, ALGOL, BASIC and Pascal have been devised. These languages use 'keywords', each of which corresponds to a long program in machine code, to put together what would be a very long and difficult program if it were written directly in machine code. When the language is interpreted, meaning that each keyword (such as PRINT, INPUT, GOTO) is dealt with by the computer in turn as the program runs, this causes the program to run much slower than if the program is compiled. Compiling means that the entire program is turned into machine code before the program is run, so that the conversion is done once and once only.

There are two particularly important reasons for using machine code. One is speed. When a computer uses an interpreted high-level language, the speed of interpretation may be such that actions like the animation of graphics or the sorting of string arrays take place much too slowly because of the interpretation process. For example, a loop such as:

```
10 FOR n = 1 TO 10
20 PRINT a
30 NEXT n
```

will require the interpretation of the keyword PRINT to be carried out ten times, and the address of variable a to be found ten times. On the other hand, a compiler would collect the PRINT and variable-finder routines, put the address of the variable into the PRINT routine, and create a loop in machine code which would carry out the action (not the creation of code, followed by part of the action) ten times. When no compiler exists for a machine, which is true of many small computers (though not of the TRS-80 or Video Genie, which can use the ACCEL compiler) this speed-up action must be carried out by writing the program directly in machine code. If the machine code routines of the interpreter exist in ROM, then it may be possible to make use of them, but the process of interpretation, finding the routine which corresponds to the word PRINT, for example, is cut out, which means that the time-consuming part of the process is eliminated.

The other reason for resorting to machine code is that any high level language is restricted by its keywords. The keywords of a language form a type of menu from which the programmer can choose, but inevitably this menu is restricted. There will be actions which are either impossible or which can only be carried out in a very clumsy way using the keywords. The most important actions of this type are inputs and outputs — the computer is generally restricted to a few standard methods, and when experimental interfacing methods are used, there will very often be no keyword which corresponds to the action that is required. It is rather unusual, to say the least, to find that computer X can read cassettes that were recorded by computer Y, and yet the methods of recording are essentially similar; only the machine code routines differ.

For every part of the system that is under the control of the mircroprocessor, however, control by machine code is possible. This aspect of machine code has assumed less importance in recent years because more recent designs of computers have paid more attention to inputs and outputs, and the requirement for specialised machine code routines is less than it once was. Nevertheless, the use of robot arms, modems, networks and other forms of interfacing will always demand some use of machine code to ensure correct operation.

The Z-80 system

The Z-80 is the most widely used 8-bit microprocessor. Eight-bit in this sense means that program or data signals sent to the microprocessor use eight voltages applied to eight corresponding pins on the body of the microprocessor. The total number of pins on the body is kept to a reasonable number (40) by using these eight pins, the data pins, both for

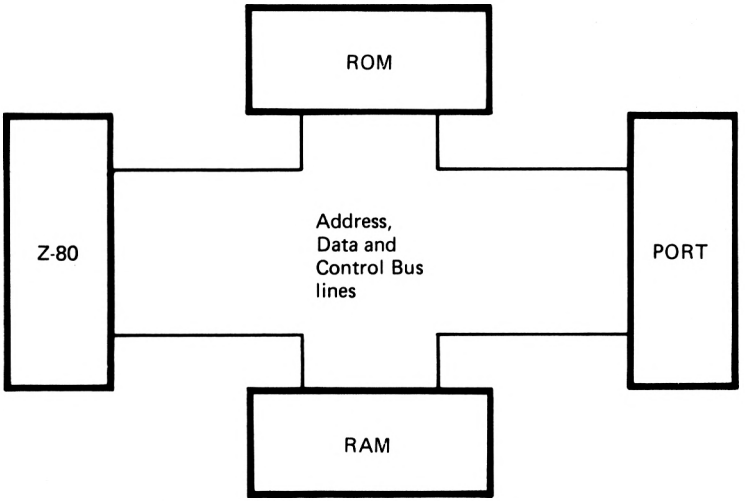


Fig. 1.1. A block diagram for any Z-80 microprocessor system

inputs and for outputs. This is possible because the microprocessor deals with actions in sequence, one by one, so that simultaneous input and output is impossible in any case.

Figure 1.1 shows a generalised block diagram of a Z-80 microprocessor system. The blocks marked ROM and RAM are memory. The ROM will contain at least the essential input and output programs that enable the Z-80 to be further programmed, and the RAM will be used to store programs or data which will be lost when the power is switched off. For most of our purposes, we shall be concerned with RAM into which the number codes that make up a machine-code program are placed, but the ROM is even more important because without some programs present in ROM it would be impossible to program the RAM. The routines in the ROM will therefore, at the very least, provide for input from the keyboard and output to the video screen, so that we can enter items and also see the effect. These inputs and outputs are carried out by passing signals through the block marked 'Port'. The port, which may be one single chip on the board or a number of connected chips, provides for connections, isolation and timing. The connections will be inputs and outputs, and these will have to be isolated from the microprocessor itself because they can be dealt with only at certain times, fixed by the program that controls the inputs and outputs. A port usually provides for storing one set of signals until they can be used either by the microprocessor or by the other systems (peripherals) that are attached to the port.

All the sections of the microprocessor system are connected by lines which are grouped as buses. The significance of a bus is that a large number of chips are made to share one common set of interconnections. The difference between independent connections and buses is illustrated in Fig. 1.2. A bus allows signals to be sent between any two units that are connected to the bus by using a common path, rather than by having a separate path for each possible signal route.

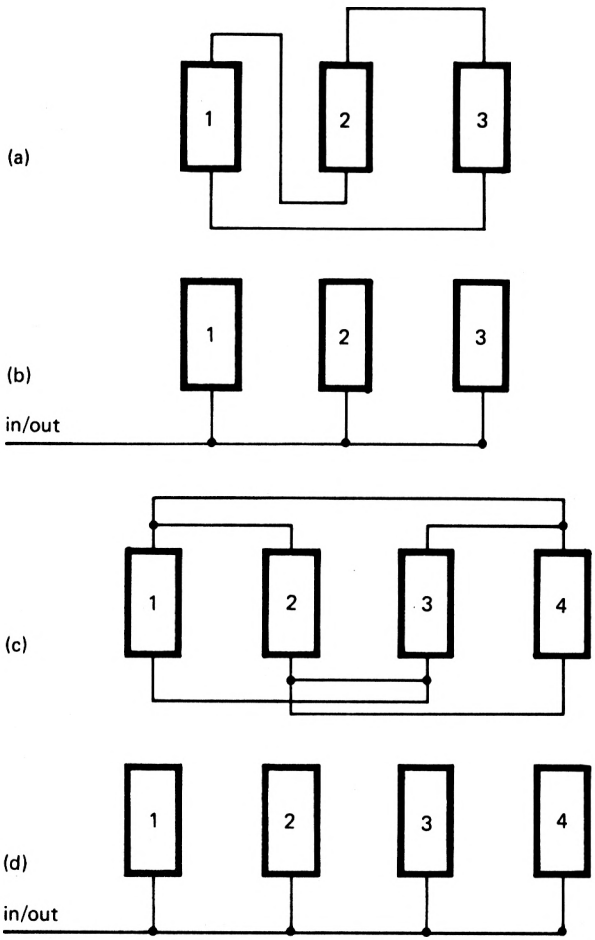


Fig. 1.2. Independent connections and bus connections. Connecting each unit of a system to each other independently (a,c) requires a large number of circuit paths. A bus method (b,d) is simpler and easier to use, provided that two signals are not sent at the same time

Words such as 'sent' and 'path' tend to lead us to think of a set of signals as something being moved from one place to another. The signals that are used in microprocessor systems, however, are electrical voltages, and a better description of their action would be 'sharing'. When a connecting line is used by any unit of a microprocessor system to 'send' a signal the electrical voltage of each line in the bus is set to one of two possible levels. Any other unit connected to the same lines can sense these voltage levels, but the voltages are not in any way removed from the sending unit and shifted to the receiving unit, any more than playing a record takes the groove from the disc and puts it into your ear! The value of a bus system is that the interconnections can be permanent, but the units are controlled so that when one unit places voltages on the lines of the bus, all of the other units on that bus will have the same set of voltage available — but they can be controlled so that only one (or more) selected units can use these voltages.

The Z-80 buses

The Z-80 buses are described as the data bus, the address bus and the control bus. The data bus consists of eight connecting lines which are bidirectional, meaning that they can carry signals from the microprocessor to other units or from the other units to the microprocessor. As we noted earlier, inputs and outputs can never be simultaneous, so that this use of the same set of connections for both purposes is not a disadvantage; it is in fact a considerable advantage in terms of the reduction of the number of connections that have to be made.

The address bus consists of 16 lines which are used for selecting units of memory. Every different arrangement of voltages on the address lines corresponds to a different portion of the memory being activated and connected to the data lines. This allows the microprocessor to 'write' — place a signal pattern on the data lines to be copied into the selected part of memory, or 'read' — when the signal placed on the data lines by the memory location is copied by the microprocessor.

The control bus consists of lines which are used to control the units attached to the data and address buses, or to allow the microprocessor to be controlled by external signals. Two important control bus lines of the Z-80 are the read and the write lines, which are connected to all the memory chips, and which send out signals from the microprocessor that will determine whether the selected part of memory is to be written to or read from. Writing implies that the content of the memory will be changed; reading always leaves the memory unchanged. Another control line is used to signal when the address bus is free to accept a

memory address; this signal is used to distinguish between the use of the address bus for memory and its use for the port or ports.

Among the control inputs are the interrupts and the reset. The two interrupt inputs allow the action of the microprocessor to be interrupted by signals which will then force the microprocessor to carry out a special piece of program whose starting address will be specified in some way. More details of this process are given in Chapter 7. The reset input, as the name suggests, clears all the data from the microprocessor, and forces the address bus lines to the number zero (that is, the signals on the lines are all at zero voltage).

In general, the signals that are present at the pins of the Z-80 are of less interest to the assembly language programmer than to the hardware designer but, since many of the hardware actions depend on software and vice versa, the hardware actions are dealt with in rather more detail in Chapter 2.

Binary number system

The signals which are present on the buses use only two voltage levels, known as logic levels 0 and 1 — the words 'logic level' often being omitted. Level 0 means a low voltage, somewhere between 0 and + 0.8 V, and level 1 means a higher voltage, between 3.5 V and 5.0 V. The wide tolerances of voltage at each level should ensure that no voltage levels in the system are ever ambiguous; all should be either at level 0 or at level 1. The use of two voltage levels leads naturally to the use of a numbering system which has two digits only, 0 and 1. This numbering system is the binary code.

Our conventional (denary) number system is based on digits 0 to 9, with the next whole number being written as 10, meaning one ten and zero units. Similarly, the number 365 means 3 hundreds (100, or 10^2) plus 6 tens (10, or 10^1) plus 5 units (1, or 10^0). Each *place* in the number shows how many units are to be counted and also the multiplier (1, 10, 100, etc.). This system of writing numbers is one of the many contributions of the ancient Arab world to mathematics and science, and it rapidly superseded the clumsy and arithmetically useless Roman number system. By using the position of each digit to indicate its power of ten, its significance or importance in the number is indicated. The '5' in 365 is the least significant digit; change it to 6 or to 4 and the change is of one part in 365, almost negligible. The '3' in 365 is the most significant digit; change it to 2 or to 4 and the change is of one hundred parts in 365 — certainly not negligible. Binary numbers can be written in the same form.

When we count using only the digits 0 and 1, however, the next

Place:	7	6	5	4	3	2	1	0
Value:	128	64	32	16	8	4	2	1

Fig. 1.3. Position values, or powers of two

number following 1 has to be 10, meaning 1 two and 0 units. The place of a digit now indicates the power of two rather than the power of ten; Fig. 1.3 shows a table of powers of two up to 2^7, embracing all the powers that will be used in Z-80 programming. Any number that can be written in denary (scale of ten) can also be written in binary (scale of two), but the binary version will contain more digits than the denary version if the number is greater than 1! The sequence of the first 16 numbers in a binary count is shown in Fig. 1.4.

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Fig. 1.4. The first sixteen numbers (0 to 15) of a binary count

Conversion of a binary number into its denary equivalent is straightforward, using the table of powers of two shown in Fig. 1.3. Wherever there is a 1 in the binary number, its place value (the value of that power of two) is written down, and these numbers are then added to give the denary number. Conventionally, the Z-80 works with sets of eight binary digits, and this set is called a byte. One byte can range from 00000000 (zero) to 11111111 (denary 255), so that the numbers that the Z-80 handles can be in sets of this size range. This does not preclude the use of whole numbers greater than 255, nor of fractions and negative numbers, provided that they can be represented in byte form, of which more later. The important point is that one byte is the maximum that the Z-80 can read in or write out at one time. Figure 1.5 shows the ancient and useful method of converting a denary number into binary form.

Using single-byte units means that special provisions have to be made for numbers that are not positive whole numbers (integers). Since the binary system makes no provision for anything other than the digits 0

Method: Divide the number by two. Write the result and the remainder, which must be 1 or 0. Repeat with the result, until the last remainder, which is 1, is found. Write the remainders in order, starting with the last one.
This is the binary number equivalent to the denary number.

Example: 124

2)124	remainders
62	0
31	0
15	1
7	1
3	1
1	1
0	1

Binary number is 1111100 (read upwards)
In eight-bit form, this is 01111100

Fig. 1.5. Converting a denary number to binary

and 1, with no + or − signs, the sign of a number, positive or negative, has to be indicated by making use of one of the bits of a byte. The bit used is the most significant bit, and the meaning allocated to it is that, if this bit is 0, the number is taken as being positive, and if this bit is 1, the number *may* be taken as negative. Note *may* — if we decide that the most significant bit will not be used as a sign bit in our programs, then we are free to do so.

Signed numbers

The use of the most significant bit as a sign bit follows from the method of creating the negative equivalent of a positive binary number. The procedure is shown in detail in Fig. 1.6; the binary number is inverted first, writing a 1 for each 0 and a 0 for each 1 in the number. Following inversion, 1 is added so as to produce the negative form, known as the 'twos complement'. This is always a number whose most significant bit

Write the binary number, single byte.
Invert each byte, writing 1 for 0 and 0 for 1.
Add 1 to the result. This is then the negative form of the original number.

Example:

Byte is	01101101	
Invert	10010010	
Add 1	10010011	this is the negative form

Fig. 1.6. Creating the negative form of a binary number. The number must use seven bits only, if its negative form is to take up one byte

must be 1, and this, or any other number in which the most significant bit (msb) is used as a sign bit, is called a 'signed binary number'. For single-byte numbers, the range of values that a signed number can have is −128 to + 127, as distinct from the 0 to 255 range of the same bytes considered as unsigned numbers.

The sequence of a binary countdown from positive values through zero to negative values looks rather odd in consequence of the use of twos complement; a portion of such a countdown is illustrated in Fig. 1.7.

11111011	−5
11111100	−4
11111101	−3
11111110	−2
11111111	−1
00000000	0
00000001	+1
00000010	+2
00000011	+3
00000100	+4
00000101	+5

Fig. 1.7. Part of a binary countdown with its denary equivalent

The important part is that the negative form of a binary number bears little resemblance to the positive form; + 5 in binary is 00000101 and −5 is 11111011. This system has, however, the very considerable advantage of permitting the microprocessor to use the same circuits to carry out both addition and subtraction. Novice programmers are often perplexed by a number such as 10011011, which if regarded as unsigned is 155 denary but which if regarded as signed is 101 denary. The curious point is that only very seldom is it important to distinguish between the meanings unless you are designing mathematical programs, because the microprocessor will treat the number in the same way no matter how you think of it. In practice, the programmer has to be aware of when the microprocessor will treat a number as being signed and when it will treat the number as being unsigned. Since we do not generally use binary numbers in the course of our programming, the number of times that the programmer is involved is negligible.

Integers and floating-point numbers

Most computers make provision for integers, meaning positive whole numbers. These are used, for example, for line numbers (except by the Camputers LYNX) and for address numbers, and they consist of two bytes each. The range of numbers is 0 to (256 × 256)−1, which is 65 535; alternatively if the numbers are thought of as signed, the range is −32 768 to + 32 767. A greater range can be obtained by using a larger number of bytes.

Numbers which are not defined as integers are treated as floating-point numbers, which are stored in mantissa-exponent form. This is analogous to the 'standard' or 'scientific' notation that is used by most calculators, in which a number is expressed in denary as a value between 0 and 10 multiplied by a integer power of ten. Numbers greater than 1 will use a zero or positive power of ten, numbers less than 1 will use a negative power of ten. Using this scheme, we can write 261 700 as 2.617×10^5, and 0.00114 as 1.14×10^{-4}. If you are unfamiliar with this method of writing denary numbers, then skip the next section — it's not for you!

Binary floating-point numbers are written in the form .XXXXXXXX for the modulus (eight bits following a binary point in this example) and with a separate exponent, the size of the power of two.

1. If number is greater than 1, divide by the next higher power of 2
2. Write in exponent – mantissa form.
3. Convert mantissa to binary fraction by subtracting negative powers of two.

Example: 24 which is greater than 1, so divide by next higher power of 2, which is 32. This gives 24 as 0.75×2^5
Now $2^{-1} = 1/2$, which is 0.5
0.75 is greater than 0.5, so write .1B, and subtract to get 0.25
$2^{-2} = 1/4 = 0.25$; in binary this is .01B, and subtracting from the 0.25 that remained leaves zero. This ends the conversion.
The fraction 0.75 denary is .11 binary.
The complete number has a mantissa of .11000000 and exponent of 10000101

Fig. 1.8. The mantissa-exponent form for a floating-point number

The scheme which is followed is that of expressing the number as a modulus whose value lies between .10000000 and .11111111 and an exponent whose value is 10000000 added to the actual value of the exponent (all figures in binary). The binary fraction is obtained by the conversion of the denary fraction to binary, which is illustrated in Fig. 1.8, having first put the number into appropriate form by dividing it by the next higher power of 2. For example, the number 56.6 is put into binary fraction form by first dividing by the next higher power of 2, which is 64. This gives the number 56.6 as being equal to $(56.6/64) \times 2^6$ (since 64 is 2^6). The fraction, 56.6/64 is 0.884375 in denary fractions, so that the mantissa of this number would be the binary equivalent of 0.884375 denary.

Looking at another example, the denary fraction 0.0246 can be converted by dividing by 2^{-5} (which is 0.03125), so that the form of the number is a mantissa of 0.0246/0.03125, in denary, and an exponent of 2^{-5}.

The exponent number, 6 in the first example, would be added to 128

(denary) to give 134 denary, in the second case, 128−5 is 123. These exponents in binary therefore become 10000110 for the 56.6 example, and 01111010 for the 0.0246 example. This allows the system to distinguish between the numbers which started greater than 1 and these which started as less than 1.

The fractional part of the number which in denary is always between 0.5 and 1 can be expressed as a number of bytes of binary. The greater the number of bytes that is used to express this fraction, the more precise the arithmetic, because denary fractions seldom convert exactly to binary fractions. Binary fractions are exact only when the denary number that is being converted is a negative power of two (Fig. 1.9) like

Power of two	*Denary*	*Binary*
−1	1/2	.1
−2	1/4	.01
−3	1/8	.001
−4	1/16	.0001
−5	1/32	.00001
−6	1/64	.000001
−7	1/128	.0000001
−8	1/256	.00000001

Fig. 1.9. Table of negative powers of two

2^{-1} (which is $\frac{1}{2}$, 0.5), 2^{-2} ($\frac{1}{4}$, 0.25), 2^{-3} ($\frac{1}{8}$, 0.125) and so on. All other numbers convert inexactly, even when four bytes are used to express the fraction as is normal in modern computers. This can lead to errors in the course of arithmetic when numbers are converted to binary fractions and back again. We encounter a similar situation when we work with numbers such as $\frac{1}{3}$ in decimal. By using four bytes to hold the fractional part of the number, however, the accuracy should be to enough places of decimals (in the denary equivalent) for all but the most stringent purposes. To avoid mistakes, though, it is essential to round off numbers to the number of decimal places that is needed.

Binary-coded decimal (BCD)

BCD, meaning binary-coded decimal (or denary), is a method of coding denary numbers using binary codes but without converting the complete number to binary form. BCD is used extensively when numbers have to be displayed on seven-segment displays, because each unit of such a display is used for one digit. Since a denary digit can range in value from 0 to 9, binary 0000 to 1001, a set of four binary digits is needed to represent each single denary digit.

A number such as 255 is represented in BCD by the binary codes for

each of its digits, giving 0010 0101 0101, rather than the true 8-bit binary equivalent, which is 11111111. The extra number of digits used, and the difficulty of carrying out calculations with numbers in BCD form, have been a barrier to the use of BCD in arithmetic routines, but the Z-80 permits numbers to be worked in BCD by the use of a 'decimal adjust' command.

Hexadecimal scale

For the purpose of programming, the most useful scale of numbering is the hexadecimal scale, known universally as hex, because it is simply related to the binary scale. The hex scale uses 16 digits, with the digits 0 to 9 supplemented by the letters A to F which are used as symbols for the denary numbers 10 to 15. The hex scale from 0 to 16, with denary and binary equivalents, is illustrated in Fig. 1.10.

Hex	*Denary*	*Binary*
00	0	0000
01	1	0001
02	2	0010
03	3	0011
04	4	0100
05	5	0101
06	6	0110
07	7	0111
08	8	1000
09	9	1001
0A	10	1010
0B	11	1011
0C	12	1100
0D	13	1101
0E	14	1110
0F	15	1111
10	16	10000

Fig. 1.10. The hexadecimal scale (hex) with its denary and binary equivalents

The hex scale is compact, requiring only two digits to represent a single-byte number, and four digits to represent a double-byte number (sometimes called a 'word'). For this reason, hex is used almost exclusively in machine-code programming of microprocessors, though denary numbers sometimes have to be used when a Z-80 is programmed in a computer that makes no provision for the use of hex.

One of the main advantages of using hex code is easy conversion between hex and binary, as is illustrated in Fig. 1.11. This easy conversion allows the programmer who uses hex to convert at any time to the binary equivalent to see what is happening to the individual bits of

Hex to binary : For each hex character, write the binary code from the table in Fig. 1.10
Example: 3AH becomes 00111010 ; E6H becomes 11100110
Binary to hex: Group the binary bits into fours, starting at the right-hand side, then substitute from the table in Fig. 1.10
Example: 01001101 is 4DH ; 1100101100 is 32CH

Fig. 1.11. Hex-to-binary and binary-to-hex conversions

a byte, and this can often lead to more efficient programming. For example, the ASCII codes for the lower case (small) letters range from 61 to 7A in hex. In binary, these are 01100001 to 01111010. The upper case letters (capitals) have ASCII codes which range from 41 to 5A in hex; in binary 01000001 to 01011010. The difference between a lower case letter and an upper case letter is simply that bit 5 (counting from the right-hand side, with the least significant bit called bit 0) is set to 1 for lower case, and reset to 0 for upper case. Detecting whether a letter is upper or lower case therefore amounts to deciding whether bit 5 is set or reset.

Conversion between hex and denary is not quite so simple. A hex number is converted to denary piece by piece, starting with the least significant digit. This is converted to denary (e.g. A to 10, F to 15) as needed, and the next more significant digit is also converted and then multiplied by 16, because its place in the number gives it a significance 16 times that of the least significant (units) digit. The next digit (if there is one) is similarly converted and then multiplied by 16 × 16, which is 256; the next digit is multiplied by 16 × 16 × 16, which is 4096. After conversion the denary results of these conversions are then added to give the denary equivalent of the whole hex number. Examples for single- and double-byte numbers are shown in Fig. 1.12.

Conversion in the other direction can be done by using successive division by 16. Dividing a number by 16 will generally result in a quotient which consists of a whole number part (integer) and a denary

Single byte:

A9

lsd is 9 units
A = 10D, which is 10 X 16 = 160 units
Total = 169 denary

Formula is: lsd + 16 X msd

Double byte:

3AF6			
	lsd is 6 units	6	
then F = 15D,	value 15 X 16	240	
then A = 10D,	value 10 X 256	2560	
then 3	value 3 X 4096	12288	
	Total =	15094	denary

Formula is: lsd + (16 X digit 1) + (256 X digit 2) + (4096 X digit 3)

Fig. 1.12. Converting hex numbers to their denary equivalents

Single byte:

119	Divide by 16, gives	7.4375
	msd is 7,	lsd is .4375 × 16 = 7
	∴ Number is 77H	

Double byte:

26420	Divide by 16, gives	1651.25		
	Fraction × 16 is	.25 × 16	=	4
1651	Divide by 16, gives	103.1875		
	Fraction × 16 is	.1875 × 16	=	3
103	Divide by 16, gives	6.4375		
	Fraction × 16 is	.4375 × 16	=	7
∴ Number is 6734H				

Fig. 1.13. Converting denary numbers to their hex equivalents

fraction. The fractional part is separated off and multiplied by 16, to get the least significant hex digit, which must be put into hex form (10 to A, 15 to F and so on). The integer part is then divided by 16 again, and the process repeated until the last integer quotient, which will be the most significant figure of the hex number, is less than 16 (denary). Examples of this conversion, which illustrate the process much more clearly than a description, are shown in Fig. 1.13.

Where hex numbers are used in programming, they are generally followed by the suffix H to denote hex, because a number such as 15 could be fifteen denary or one-five hex, which would be 21 denary. Throughout this book all numbers will be written in hex unless denary or binary is specifically stated. Where the point must be emphasised, or where confusion is possible, the H suffix will be used to indicate hex and D to mean denary.

2

Z-80 Architecture

Figure 2.1 illustrates the architecture of the Z-80 from a programmer's, as distinct from a hardware designer's, point of view. A programmer is interested primarily in the registers of the microprocessor, which are memory blocks within the microprocessor whose actions can be

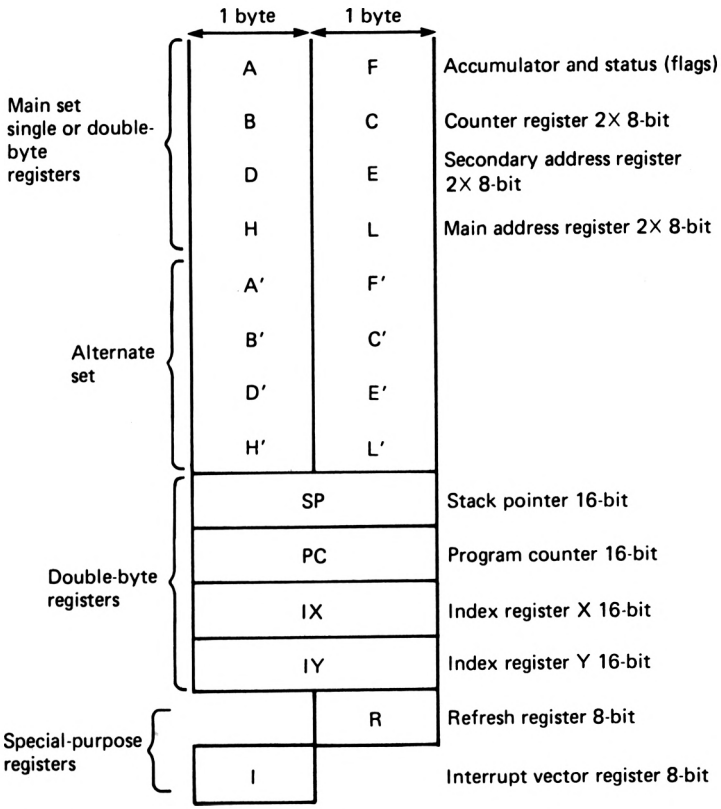


Fig. 2.1. The registers of the Z-80. This shows the registers that can be programmed by the user — there are several registers which are not accessible to the programmer

controlled by electrical signals at the inputs. As compared with other microprocessors of the same design period, the Z-80 is notable for:

1. The very large number of registers that can be used by the programmer;
2. The use of many of the registers singly as 8-bit registers or in pairs as 16-bit registers;
3. The very large instruction set (number of possible instructions);
4. The provision for storing the contents of registers in pairs into the memory (stack) of the microprocessor system at addresses chosen by the programmer;
5. The number of 'block shift and compare' instructions, each of which can take the place of a whole sequence of single instructions;
6. The provision on the chip for supporting refresh signals for dynamic memory;
7. The very large number of control signals, which makes it easy to interface Z-80 systems with other systems.

These features make the Z-80 a particularly fascinating chip to program, but the huge instruction set of 697 codes makes it intimidating for the beginner. Many of the commands of the Z-80 are identical to those of the older 8080, for which the Z-80 was designed as a replacement, and this causes some duplication of commands. There are, for example, several 8080 commands within the Z-80 instruction set which are not needed by a Z-80 programmer because improved Z-80 commands are available to carry out the same tasks. Unless the beginner is aware of this, a lot of time can be wasted on trying to unravel the reason for having apparently similar instructions. This book will concentrate on the more useful and the most commonly used items of the instruction set.

Hardware signals

The usefulness of a microprocessor lies in the range of signals that it can accept as inputs or produce as outputs, so we shall start our examination of the Z-80 by looking at these signals, starting with the data and the address lines.

The Z-80 uses the conventional eight data lines and 16 address lines of any 8-bit microprocessor. Both of these sets of lines are three-state, meaning that the pins on the casing of the Z-80 can be isolated from the processing circuits inside until the lines are needed. The data lines are also bidirectional, meaning that signals can be passed to or from the Z-80 along the same set of lines. This is no handicap because the microprocessor deals with one set of signals at a time, strictly in sequence, and would never be using the lines for both purposes. The

address lines are output-only lines, carrying signals that are used to select memory. The timing of the sequence of actions of the microprocessor is determined by an external 'clock' oscillator. The maximum oscillator frequency for the Z-80 is 2 MHz; the Z-80A permits 4 MHz operation, and the Z-80B permits 6 MHz operation. The features that distinguish the Z-80 from so many of the other chips designed at or around the same time, however, are the number and the variety of the control signals that are available.

Referring to the pinout diagram of Fig. 2.2, pin 27 is used for the M1 output, which is normally high (logic 1) and is active when low (logic 0). This output goes low to signal to the rest of the system that the microprocessor is on the fetch part of an instruction (see later), meaning that it is about to access memory. This signal, like several others, is used to synchronise the action of other circuits to the actions of the microprocessor and, also like several others, is not necessarily of use in all types of circuits.

The signal output on pin 19 is labelled MREQ. This output is normally isolated, being one of the three-state outputs, but it goes to logic 0 when it is active to indicate that the voltages on the address pins can be interpreted and used as a valid address or for memory reading or

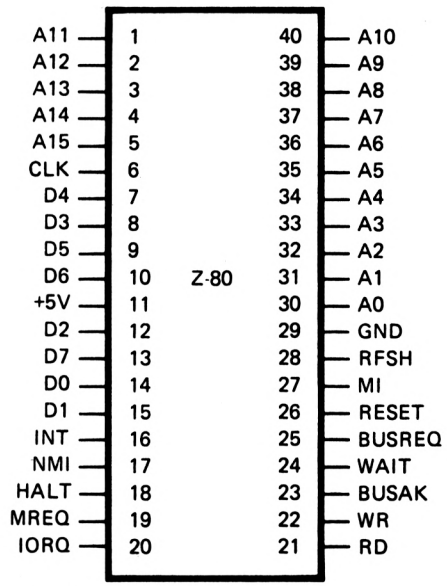


Fig. 2.2. A pinout diagram for the Z-80, showing the abbreviations for the signals at each pin

writing. This signal is seldom used on its own; it is more commonly gated with the memory read or write signals to ensure that memory is not activated by undesirable signals and that port inputs and outputs are separated from memory inputs and outputs.

Another three-state output is found at pin 20, the output labelled IORQ. This is used in association with port actions, and goes low when the lower eight bits of the address lines (the address bus lower byte) are being used for a port input or output. The IORQ abbreviation means Input/Output Request, but its output will also go low to acknowledge an interrupt (see later), so that the signals from this pin will have to be gated with other signals to separate the two uses.

The RD (read) signal on pin 21 goes from its isolated state to low when the Z-80 is about to start a data read, either from memory or from a port. Another similar action, WR (write) at pin 22, takes place to indicate that a write action has started, sending data signals to a memory or to a port. These two signals will have to be used by the memory chips of the microprocessor system, usually with gating or a flip-flop added because most memory chip systems use a single RD/$\overline{\text{WR}}$ pin, logic 1 to read, logic 0 to write, rather than the two-signal arrangement of the Z-80. An R-S flip-flop, with the RD and WR signals as inputs, and with pull-up resistors to ensure that the logic level at each input is 1 when the outputs are isolated, is usually adequate.

The output at pin 18 is HALT — an output which goes low when the Z-80 encounters a software HALT instruction. The effect of this is to stop processing until an interrupt signal or a reset signal is received. The buses are still active during this period, however, because refresh signals are sent so as to maintain dynamic memory. The output on pin 28 is the RFSH (refresh) output which goes low to indicate that the lower byte of the address bus is being used for memory refresh signals. These signals will not be needed if static memory is in use.

Of the inputs, BUSREQ (bus request) on pin 25 is normally high and must be taken low to be active. When this signal is received, then, at the end of an instruction, the address bus, data bus, and control outputs will be placed in their isolated states, allowing other devices (a second Z-80 perhaps) to take control of these lines. This encourages the design of systems in which different actions are allocated to different microprocessors — a type of system which is becoming increasingly favoured for computers. In this one way, one microprocessor can handle one class of processes, and another can handle others, each working independently until the time comes to transfer information between the systems. There is a corresponding output, BUSAK on pin 23, which goes low to acknowledge that the buses are isolated, and whose action is so bound up with that of BUSREQ that I have listed it here rather than with the outputs.

The WAIT input on pin 24, when taken low, will cause the Z-80 to halt processing, except for memory refresh, in much the same way as a HALT software instruction. This is a useful way of forcing the Z-80 to go through its operations one step at a time, and can be a useful facility when programs are being developed (the use of single stepping is dealt with in Chapter 8).

The RESET input on pin 26 will, as the name suggests, clear the internal register memories of the Z-80, ready to run a program. The address bus output is reset to zero, interrupts are enabled, and all other registers cleared. The address and data buses are isolated and all control inputs are put into the inactive (isolated) state. A reset will therefore be followed by whatever instruction is stored in memory at address ØØØØH, which is where all programs for the Z-80 must have a start-up instruction placed (which may simply transfer control to another address). The use of RESET has no effect on memory: memory is never cleared by any action of the microprocessor other than when running a program written for the purpose of clearing memory.

Two inputs are reserved for interrupts, which are actions that cause the microprocessor to suspend its normal sequence of actions so as to carry out other actions which have a higher priority. The INT input on pin 16 will be active when its voltage is taken low. This will have an effect only if the software instruction 'interrupt enable' has been performed previously, or a RESET has enabled interrupts. Another software instruction, 'disable interrupts' (DI), allows the programmer to overrule this interrupt input so that operations which depend on critical timing, such as the transfer of information to and from disc, cannot be interrupted.

The second interrupt system cannot be overridden — it is referred to as a 'non-maskable' interrupt, NMI. Pin 17, when taken low, will force an interrupt at the end of the instruction which is being performed, and will cause the address lines to put out the address ØØ66H. It is, as usual, the responsibility of the programmer to make sure that instructions are present starting at this memory address that will take care of the interrupt. Details of this process and of the program which is needed are given in Chapter 7.

Registers

A register is a single block of storage for one or two bytes, similar in construction to the circuits used for static memory, but unlike the memory of a microprocessor system a register can be controlled by other units called gates. The microprocessor can manipulate bits in a register and each manipulation is controlled by these gates, using the instruction

received from memory as a 'template' for the settings of the gates. The general pattern of any microprocessor action is that a byte is copied from some location in memory to a register in the microprocessor. It is then processed and can be returned to the memory, either to the original address location or to another one. The register arrangement of a microprocessor therefore determines how an action can be programmed, so that programming a microprocessor demands a fairly detailed knowledge of the structure of that microprocessor. This contrasts with programming in higher-level languages like BASIC, for which virtually no knowledge of the structure of the machine is needed to be able to program effectively. Give or take a few quirks, if you can program a TRS-80, for example, you can program a ZX Spectrum, or a Dragon, or a Commodore 64, or a Lynx, but being able to program a 6502 microprocessor doesn't make you skilful in programming a Z-80, though you start with some advantages as compared to a novice.

The single registers

The Z-80 contains a number of single-byte registers, meaning registers that can store and work with one byte at a time. Some of these single registers can be used in pairs as double registers (register pairs), but for the moment we shall concentrate on the one-byte actions, which are dealt with in much more detail in Chapter 5.

Of the single-byte registers, the most important is the accumulator, or A-register. It is important because:

(a) there are more instructions that affect the A-register than any other;
(b) the results of actions that involve the A-register and any other register will be stored finally in the A-register alone; and
(c) it is easier to transfer bytes between the A-register and memory than between any other register and memory.

The accumulator is therefore the main 'action' register into which a byte will be placed to be processed. The name 'accumulator' is derived from the fact that the result of any operation that has used the accumulator is always stored back in the accumulator. If, for example, we add a byte taken from some memory address to another byte stored in the accumulator, then the result of the addition will be stored in the accumulator, replacing the byte which was previously stored there. We could simulate the action of this register in BASIC by having a variable A which would be used in statements like:

```
100 LET A = A + byte
```

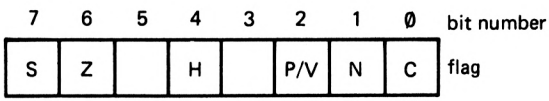
where 'byte' is a number variable which is being added to the value of A.

Since the variable A accumulates in value each time this type of instruction is used, the name accumulator was chosen for the register whose action is so similar. Note, however, that the A-register cannot hold more than one byte, so that the value of this byte cannot accumulate indefinitely; its maximum value is FFH.

Many microprocessor types rely on one accumulator, perhaps two, but the Z-80 has a collection of six other single-byte registers which can be used in some of the same ways as the accumulator, but without the accumulating action. These are single-byte registers that are used mainly as feeders to the accumulator, holding or manipulating bytes that are to be used later in conjunction with a byte stored in the accumulator, with the result being placed in the accumulator. These other single-byte registers labelled as B, C, D, E, H and L can also be connected internally in pairs, a topic we shall examine later.

Two other single-byte registers, the I and R registers, need not concern us at the moment. The R-register is used to contain the lower half of an address that is used for refreshing dynamic memory. This is an automatic action, and the only time a programmer is likely to make use of the R-register is to copy the byte stored there to use as the basis for generating a random number. The I-register contains the lower part of the address for an interrupt program, and will be used only if the hardware of the microprocessor system makes use of that type of interrupt. This type will be dealt with in more detail in Chapter 7.

The other eight-bit register that is important to the programmer is not really a register at all. The status, or flag, register, labelled F, is a collection of single-bit stores rather than a store for a complete byte. Each bit is an indicator, set (to 1) or reset (to 0) by the result of an operation that has taken place, usually in the accumulator. We can trace its action with reference to the most important bits, the zero, sign and carry flags (Fig. 2.3). These flags each occupy one place in the register, and are numbered in the conventional register manner, starting with 0 for the least significant bit and going up to 7 for the most significant bit.



Bits 3 and 5 are not used. These bits may be at Ø or 1

S	Sign bit, takes same value as msb of register
Z	Zero bit. Set to 1 when a register value is zero
H	Half-carry bit. Used in BCD arithmetic
P/V	Parity/overflow bit. Used for (a) serial data checking (parity) (b) signed arithmetic (overflow)
N	Negate bit. Set when a subtraction is taking place
C	Carry bit. Set when there is a carry from msb of register

Fig. 2.3. The layout of the bits in the flag register

The zero flag, Z, is in bit 6 of the flag register, and it will be set to 1 if an action in the accumulator or another single-bit register such as B, C, D, E, H or L, has resulted in a zero result. For example, if the accumulator contained the byte 3FH, and we then subtracted 3FH from this, the result would be zero stored in the accumulator, ØØH, and this result would cause the Z-flag to be set (logic 1). Subtracting 2FH would have caused the Z-flag to remain reset (to 0), and subtracting 4FH, which would give a negative answer would leave the Z-flag reset, but would cause both the sign flag (bit 7) and the carry flag (bit 0) to be set to 1. An operation which leaves the most significant bit of the A-register at 0 will also leave the sign flag at 0; if the msb of the A-register is 1, then the sign flag bit will also be 1. In this way, if numbers are regarded as signed, then the sign bit in the flag register will keep track of the sign.

```
          00110100
        + 00011011
carry   ----------
0       01001111
        ----------
result fits in 8 bits, carry bit 0

          10010110
        + 11011001
carry   ----------
1       01101111
        ----------
result needs 9 bits, carry bit 1
```

Rules: 0 + 0 = 0
0 + 1 } = 1
1 + 0 }
1 + 1 = 0 + carry 1
1 + 1 + 1 = 1 + carry 1

Fig. 2.4. How the carry bit is used as a ninth bit for the accumulator

The carry bit is used as a ninth bit for the accumulator. When two single bytes are added (Fig. 2.4) the result may need nine bits rather than eight, so that a single-byte register cannot hold the complete result. The most significant bit, which is the carry from the msb of the accumulator, is then stored in the flag register as the carry bit. This flag will be set only if a carry has occurred from the msb of the accumulator, and is reset otherwise. The same flag is also used in a subtraction process, when a large number byte is subtracted from a smaller one thus giving a negative result. In this case, the carry bit is set and is used as a 'borrow' bit.

The importance of the flag register is that it enables the microprocessor to test the results of actions that have taken place in its registers, and in Chapter 4 we shall examine how these flags can be used to control jump instructions which execute the equivalent of the BASIC instruction:

IF A = 0 THEN GOTO NN where NN is an address number.

Two bits of the flag register, incidentally, are not used, though they may be found to have values of 0 or 1, and some of the others, such as bits 2 and 4, are of less interest to the beginner and are seldom used.

Double registers

The most important double register is the program counter, PC. This can store two bytes, and its function in the Z-80 is to store an address number. Any number held in this register can be used in the form of signals on the address bus of the microprocessor to select one byte of memory from all others. This byte of memory can then be written to or read by the microprocessor. Writing means that the contents of a microprocessor register will be copied into the selected byte of memory, replacing anything that was stored at that memory address previously, but with no change to the byte in the register. Reading means that a byte stored at the selected memory address will be copied into a register of the Z-80, replacing any byte that was previously stored in that register, but not changing the content of the memory address.

The PC is seldom directly involved in a program instruction, because its action is almost completely automatic. At the end of an instruction or the processing of a byte, the normal action of the PC is to increment, meaning that the address number stored in the PC has 1 added to it. This ensures that the normal action of the microprocessor is to select memory addresses in sequence, one after the other as required. This action can be changed momentarily by a jump instruction which causes a new address number to be placed onto the PC, but after this new memory address has been used the incrementing action resumes. A subroutine call will also cause a different (out of sequence) address number to be placed in the PC, but in this case the original number of the sequence is resumed after the subroutine has been completed. These actions are illustrated in Fig. 2.5. The main concern for the programmer is the method by which the address of the first byte of a machine-code program is placed into the PC, because this is the action that causes a machine-code program to run. This method varies from one computer to another — one common method is to use the BASIC command USR followed by an address. The topic is dealt with further in Chapter 8.

Several of the other double registers are formed by combining single registers. This is done by gating inside the microprocessor when the correct instructions are received, and the pairings are fixed as BC, DE, and HL, with B, D, and H used for the higher order byte and C, E, and L used for the lower order byte respectively. These register pairs can be used to hold and manipulate any two-byte numbers, but the instructions that are available for these registers favour the use of the BC pair as a counter, holding a number that is used for a count-down or count-up;

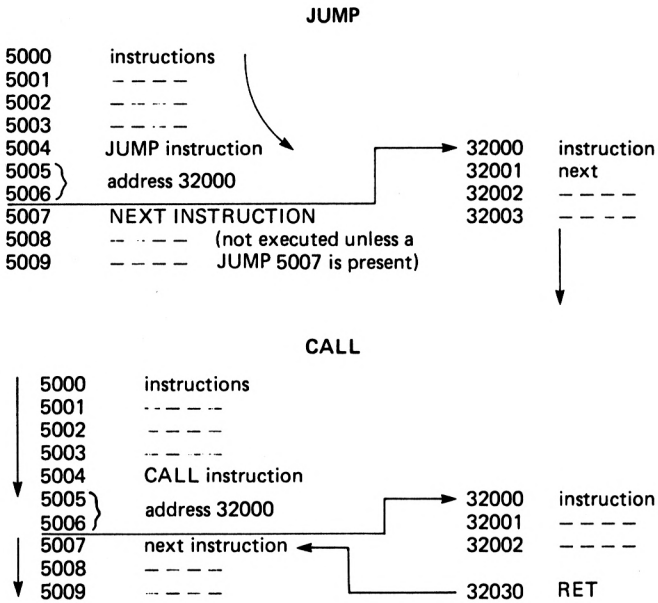


Fig. 2.5. Changing addresses by means of the JUMP and the CALL instructions

the HL and the DE register pairs are then used as spare address registers, holding addresses that are used in the course of the program. This use of double registers is a very important feature of the Z-80 and one which makes life much easier for the Z-80 programmer as compared to earlier designs of microprocessors.

Stack pointer and index registers

Three other double-byte registers are referred to as the stack pointer (SP) and index (IX and IY) registers respectively. The stack is a name given to a section of RAM memory which the microprocessor can use for the temporary storage of quantities while an instruction or a series of instructions is being carried out. Unlike the 6502, whose stack must be a section of memory whose starting (top) address is Ø1FFH, the piece of RAM that is selected for the Z-80 stack can be anywhere in the addressable RAM, and the programmer selects the starting address by placing it into the stack pointer register. The programmer can also check what piece of the stack is being used by examining the contents of this register.

When the Z-80 that you are programming is part of an existing

computer system the stack pointer address will have been determined by the ROM of the computer, and you will not normally have to change this address unless it conflicts with your intended use of memory. If you are writing a program for a control system whose hardware is not in final state then you will have to select a stack address for yourself — a frequently used address is the highest address in RAM. The use of the stack is explained in more detail in Chapter 7.

The index registers are each double-byte registers which can be loaded (filled) with an address. The address that is stored in an index register is referred to as its 'base address', and other addresses up to 127 places higher or 128 places lower can be used by adding a single byte, regarded in this instance as a signed byte, to this base address. This is used as a method of obtaining addresses for bytes in memory, and its particular advantage is that a completely different set of addresses can be obtained from the same set of instructions simply by changing the base address. For example, if you have a set of instructions that makes use of a base address stored in IX, perhaps 6FFFH, with important bytes stored at IX + 3, IX + 6 and IX + 9 (7ØØ2, 7ØØ5, 7ØØ7), then the same instructions can be used with a set of different bytes simply by changing the address stored in IX, perhaps in our example to 5FFFH. The use of two index registers, IX and IY, means that two sets of such related bytes can be referred to, and these base addresses can be interchanged with addresses held in other double-byte registers, particularly HL.

Fetch and execute

The title looks like one of Henry VIII's favourite phrases, but it is in fact the fundamental action of the microprocessor. Though a precise description of the fetch and execute cycle is of interest only to the hardware specialist, since it concerns the nature and timing of the signals at the pins of the microprocessor, an outline of the process is important for the Z-80 programmer. Each instruction begins with a fetch action, which must access memory and find an instruction byte stored at the address that is used. The microprocessor is strictly a sequential device, so that unless the bytes of the program are stored, by the programmer, in the correct order and are arranged with the correct starting byte, a program cannot run correctly.

The first byte fetched at the start of a program is therefore treated as an instruction, and before any attempt is made to carry out this instruction, the byte must be analysed. This is done in the instruction register, a single-byte register which cannot be controlled in any way by the programmer — its action is determined by a microprogram which is built into the design of the microprocessor. The first test that is carried

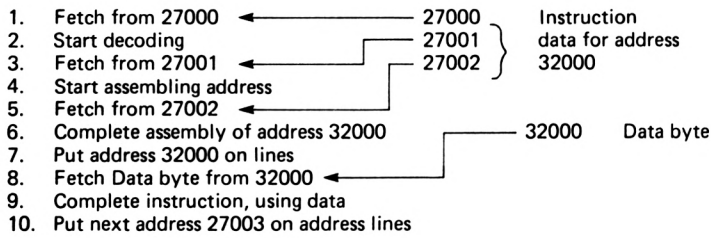


Fig. 2.6. The fetch and execute cycle illustrated

out in the instruction register is to determine if any further fetch cycles are needed to complete the instruction. It may be, for example, that the byte is one of a pair comprising a two-byte instruction, or that the instruction is one which cannot be carried out without fetching a byte of data (a number to add, for example), or two bytes to form an address. The structure of the instruction byte, meaning the way the bits are arranged, will carry this information, and if further fetch cycles are needed they will be carried out, incrementing the PC each time a byte is read. The additional bytes that are fetched in this way will be routed appropriately — if a second instruction byte is needed, then it will be routed to the instruction register, replacing the first ('pathfinder') byte. If a single byte of data is needed, then it is routed to the accumulator where it will be used, or to another register if this was specified in the instruction. If an address is read in (two bytes) these bytes have to be put together into an address register (one which is not accessible to programmers) so that the contents of this register and the contents of the PC can be interchanged to allow the address to be used, and then interchanged again to allow the PC to resume at its former address (Fig. 2.6).

When the fetch cycles have been completed and any data that is needed has been loaded in, the action of the instruction can be carried out. This is the execute part of the cycle, and it is usually the briefest part of the whole instruction. Execution is always quick compared to fetching, so when a program must run fast it is important to keep data so that it can be fetched quickly. As far as the Z-80 is concerned, storing data in the registers is the best way of ensuring rapid execution.

3

Assembly language and assemblers

A machine-code program for the Z-80 consists of a string of single-byte numbers, some of them acting as instruction codes, others as data, parts of addresses or ASCII codes, all arranged in sequence. The sequence is important because there is nothing else to distinguish one number from any other. The byte 47H may be the instruction signifying 'load register B from the accumulator', or it may be a data byte for the number 71 denary, the ASCII code for the letter G, or part of the address 7F46H. The way in which each byte is used by the Z-80 depends on the sequence of that byte in the fetch-execute cycle, and if the sequence is wrong, the action will also be incorrect. In addition, when you program in machine code the machine will not issue any error messages to you — you have bypassed its system and you are on your own!

Trying to write a machine-code program as a set of numbers is very tedious, and like any other tedious operation is error-prone. For this reason, we prefer to design and write machine code in what is called assembly language — a system which is simpler to use and much easier to follow. Each instruction is represented by a *mnemonic* — an abbreviated word which is enough to remind the programmer of the complete action — and following this mnemonic, the data byte or bytes which are associated with it are written in hex. By adopting a set of conventions for writing this language, conventions which were laid down by Zilog Inc, who designed the Z-80, it is possible to write programs in a form which other programmers can read and recognise, a task which is very much more difficult if the program is written in machine-code numbers.

Assembly language format

Assembly language consists of lines arranged with one instruction per line, like a simple BASIC program. Each complete instruction consists of two parts: the *operator,* which describes what is to be done; and the *operand,* which describes what is used to perform the action — the data on which the action is performed, for example. Writing a program in

assembly language has two enormous advantages. One is that the program becomes easier to follow. This is a comparative advantage because, like a poorly written BASIC program with no REMs, an assembly language program can be very hard indeed to follow unless you happen to know what the programmer was aiming at. Nevertheless it is very much easier to follow than its machine-code equivalent. The second advantage is that the conversion of assembly language into machine code is a purely mechanical operation of translation. It can be done 'by hand' by looking up the machine-code number(s) for each instruction but, better still, it can be carried out by a computer program which is called an assembler. When assembly language lines are typed into a computer which has been programmed with an assembler, the assembly language can be stored, recorded and replayed, and then assembled into machine code which can also be stored or recorded. Unlike hand conversion, assembly by an assembler is 100 per cent correct, and so the use of an assembler is virtually essential for any program of more than about 25 to 50 bytes, depending on your own threshold of boredom.

Opcode, operand and labels

The operators or *opcodes* of the Z-80 consist of simple two, three or four-letter mnemonics such as LD (load), ADD, AND, OR and so on. At this stage, there is little point in looking at a list of all possible Z-80 mnemonics because the list is very long, so we shall use a few as examples. In this way, you learn the style of assembly language programming, and you can then check when you come to write programs whether the command you want to use actually exists. Since LD is the opcode that is used more than any other, it will form a good basis for examples.

The opcode LD must be followed by a first section of operand which states the destination of the load. Load means that a data will be copied from one store (memory or a register) to another, and the complete operand for the LD command must state both the source and the destination of the load. The source part of the operand is indicated by another letter (if it is a register) or a hex data byte or a hex address, with a comma separating this source section of operand from the destination section.

For example, the command LD A, C means 'load the A-register from the C-register', in other words, copy the contents of the C-register into the A-register. This will leave the C-register unchanged, but will change the contents of the A-register from their previous value, making them identical to the contents of the C-register. By reversing the order of the parts of the operand we reverse the process, so that LD C, A means 'load

the C-register from the A-register'. The order of destination and source in the operand of a Z-80 instruction is very important, because it is used to replace the commands which 6502 programmers know as ST (store). Thus LD (32A6H), A means 'load the memory at address 32A6H with the byte in the A-register' — in 6502 mnemonics this would have been written as STA 32A6H. The use of brackets in Z-80 mnemonics always indicates 'contents of', so that the number 32A6H means the two-byte *address* of that number, but (32A6H) means the single byte which is *stored* at that address — this is a very important distinction.

In addition, assembly language permits the very useful device of labels. A label is a word, often limited to a length of six letters, which can be used in assembly language programs in place of an address or a data byte. We can, for example, write commands such as:

```
LD A,(SCREEN)
```

or

```
LD (PIXFIL),A
```

in which the words represent addresses, or we can have

```
LD A,EOF
```

in which the word EOF represents a data byte. Note the lack of brackets in this latter case, and also the attempt to make the label word have some meaning — EOF is taken as the End-of-File byte in cassette or disc systems.

A program which contains labels such as these will not, however, assemble correctly unless the values that the labels represent are stated in some way. One method of stating values is by a 'pseudo-instruction' called EQU, in a line of the form:

```
SCREEN EQU 3000H
```

which supplies the address number corresponding to the label name. If this line is made one of the first in the assembly language program, then the address 3000H will automatically be inserted during assembly at each occurrence of the label name SCREEN. The merit of this, as distinct from using the actual address number each time it is needed, is that in a long program, where this address may be used many times, all the addresses can be changed simply by changing one line, the EQU line. In addition, the program becomes easier to follow because the words are more memorable than the numbers — it's easier to think of SCREEN as an address for the first byte on the screen than its actual address number. The ability to change address numbers in one operation is particularly useful if you are writing code for a computer which exists in two different memory sizes, for example. The word 'pseudo-instruction' is used for EQU because the line containing EQU

does not result in any machine code being generated, it is simply an instruction to the assembler.

The other way of attaching a value to a label name is to place the label name before an instruction in a line. If we write, for example,

DESTN: LD A, 3AH (note the colon used as a separator)

then the label DESTN will indicate the address at which the first byte of the instruction LD A,3AH is placed in memory. As before, this address will be placed into the assembled machine code wherever the label word DESTN appears, and needless to say, each label word should be used for only one address in any one program.

An assembler may be a single-pass or a two-pass type. When a label word is encountered during assembly and has not been previously defined, a single pass assembler can leave a vacant space for the address, and fill in the value when the label word is encountered later. A two-pass assembler does not do this — it stores a list of all label words and corresponding addresses or bytes, and fills them in on a second run (pass) through the code. Most two-pass assemblers will need only one pass if all the label names are defined before they are used; a second pass is needed only for 'forward references', meaning that a label name is defined later in the program than its first use.

Another pseudo-instruction used by assemblers is ORG. This, as you might suspect, means the origin of the program, the address at which the first byte of the program will be assembled. If we type:

ORG 23600

or

ORG 7FF0H

then the assembler will create machine code which is intended to start at the address stated in the ORG statement. This does not mean to say that the code will actually be placed at these addresses — some assemblers simply generate code for display on the screen, or to store on disc or cassette, in which case the disc or cassette must be replayed to get the code into memory starting at the ORG address. Assemblers which place code directly into memory have to be used with some caution, because carelessness in specifying the ORG address can lead to the assembled code replacing some of the assembler program itself, resulting in a crash which generally makes the keyboard ineffective. The only remedy open to you then is to switch off or to reset; the subsequent loss of the program will remind you how important it is to save the assembly language (the source code) each time you change it, and certainly before you assemble machine code and try to run it!

Figure 3.1 shows the main Z-80 mnemonics and a brief explanation of their actions. These, a small number from the set, have been selected

ADC A	Add byte and carry bit to accumulator
ADD	Add byte to accumulator
AND	And byte with byte in accumulator
CALL	Call subroutine (address must follow)
CP	Compare with byte in accumulator and set flags
DEC	Decrement – subtract 1
INC	Increment – add 1
JR	Jump relative to PC address
LD d,s	Load to *d*estination from *s*ource
POP	Take from stack memory
PUSH	Put on to stack memory
RET	Return from subroutine
SUB	Subtract, to borrow bit

Fig. 3.1. The main Z-80 mnemonics. A very large part of the whole Z-80 instruction set consists of variations on these mnemonics

simply on the basis that they are used most frequently, and that variations on these mnemonics account for several hundred of the 697 possible Z-80 instructions.

Addressing methods

In a BASIC program, we are accustomed to using statements such as:

```
LET a = 62
```

and

```
LET b = a + 2
```

without any thought as to how and where the values that correspond to the variable names 'a' and 'b' are stored. You can't do this when you work with assembly language, because machine code demands details; each byte belongs somewhere, and if a byte is to be placed in a register you have to specify the register and where the byte is to be copied from. The different ways in which the sources of bytes can be specified are called the addressing methods for the Z-80.

The simplest possible addressing method is one which uses no source — the byte is already in a register and an operation is carried out on it. The action NEG is of this type — it converts the number stored in the accumulator into its twos complement (negative) equivalent. No memory or other register has to be specified, so no operand is needed.

Immediate addressing

A more genuine addressing method is the one known as immediate

addressing. Immediate addressing means that the data byte which is to be used will be found immediately following the instruction byte in the memory. If, for example, we use the assembly language instruction:

LD A,2ØH (note the comma between destination and source)

then the assembled machine code (Fig. 3.2) will be placed into memory in the sequence 3EH, 2ØH, where 3EH is the machine-code byte for 'load A immediate' and 2ØH is the byte which is to be loaded. Since the

Assembly language	*Memory address*	*Content*
ORG 32500D		
LDA,2ØH	325ØØ	3EH
– – – – – –	325Ø1	2ØH
	. .	. .

Fig. 3.2. How immediate-addressed data bytes are stored in the memory

(DA = data byte)

ADC A, DA
ADD A, DA
AND DA
CP DA
LD(HL), DA
LD A, DA
LD B, DA
LD C, DA
LD D, DA
LD E, DA
LD H, DA
LD L, DA
OR DA
SBC A, DA
SUB DA
XOR DA

Fig. 3.3. Instructions which can be immediate-addressed

data byte 2ØH is obtained by the normal incrementing action of the PC, this is a very fast and easy way of obtaining data, but it cannot always be used. Immediate addressing (Fig. 3.3) requires the data to be placed in the same area of memory as the program itself, and it's often much more convenient to place such data bytes together at a different address. In addition, if the program is to be put into the form of a ROM, using immediate addressing will result in fossilising data values — if you want to change the data values, then you have to change the ROM. Most designers prefer to place the instructions in ROM and most of the essential data in RAM, using a ROM routine which places fixed values of data in RAM when the machine is switched on. This way, the data is available from the ROM, but it can be changed at will and new values

used until the next time the machine is switched on. Other addressing methods which can cope with addresses that are out of the proper sequence are therefore needed.

Direct addressing

Direct addressing is one such method. When an instruction is direct addressed, a complete two-byte address has to be supplied either as a number, denary or hex, or as a label name which has been defined by EQU or by being placed before an instruction. Examples of direct addressing are:

```
LD A, (FBØØH)
```

or

```
LD A, (SOURCE)
```

using the brackets as usual to indicate that the byte stored at that address

```
CALL ADDR
CALL C,ADDR
CALL M,ADDR
CALL NC,ADDR
CALL NZ,ADDR
CALL P,ADDR
CALL PE,ADDR
CALL PO,ADDR
CALL Z,ADDR
JP ADDR
JP C,ADDR
JP M,ADDR
JP NC,ADDR
JP NZ,ADDR
JP P,ADDR
JP PE,ADDR
JP PO,ADDR
JP Z,ADDR
LD(ADDR),A
LD(ADDR),BC
LD(ADDR),DE
LD(ADDR),HL
LD(ADDR),HL
LD(ADDR),IX
LD(ADDR),IY
LD(ADDR),SP
LD A,(ADDR)
LD BC,(ADDR)
LD DE,(ADDR)
LD HL,(ADDR)
LD HL,(ADDR)
LD IX,(ADDR)
LD IY,(ADDR)
LD SP,(ADDR)
```

Fig. 3.4. Instructions that make use of direct addressing

is to be loaded. A command such as LD A,FBØØH (without brackets) should be rejected by an assembler, because a single-byte register cannot be loaded with a two-byte number.

Figure 3.4 lists the instructions which can make use of direct addressing. Note that of the single-byte registers only the accumulator can be loaded in this way — there is no instruction such as LD C,(ADDR) for example. Similarly, only the accumulator can be used to load a byte into an address using the form LD (ADDR),A. The double registers (or register pairs) BC, DE, and HL, together with the index registers IX and IY, can be loaded from memory or can load into memory using instructions of the form:

LD BC, (3CØØH) or LD (3CØØH), DE

but this form is not exactly the same as that of LD A,(ADDR). When LD BC, (3CØØH) is used, for example, the byte stored at address 3CØØH will be loaded into the C-register, and the byte stored at address 3CØ1H will be loaded into the B-register. A register-pair load therefore takes bytes from two consecutive memory addresses, of which only the first has to be specified in the instruction. The result is that the first of the two bytes is loaded into the lower register (C, E or L), and the second of the bytes is loaded into the higher register (B, D or H) when the BC, DE and HL pairs are used. When two consecutive memory addresses are used to contain another address, therefore, the address must be stored low-byte first. If, for example, we had 3FH stored at 3CØØH, and 7FH stored at 3CØ1H, then the address that would be put into HL by the command:

LD HL, (3CØØH)

would be the address 7F3FH, not 3F7FH.

Wherever a number is stored as two bytes, the Z-80 always requires the storage to be in this low-byte, high-byte order, and programmers must try to keep to this order as far as possible unless very good reasons exist for departing from it. A good example is the way that line numbers are stored in the ZX Spectrum — this is done high-byte first so that the

Command	*Time*	*Effect*	*Bytes*
LD (ADDR), A	13	Byte in A → ADDR	3
LD (ADDR), BC	20	Bytes in BC → ADDR and ADDR+1	4
LD (BC), A	7	Byte in A to BC address	1
LD (IX + dis), A	19	Byte in A to IX + dis address	3
LD A, B	4	Copy byte from B to A	1
LD A, DA	7	Load A immediate	2
LD BC, ADDR	10	Load BC with 2-byte address	3
LD IX, ADDR	14	Load IX with 2-byte address	4

Fig. 3.5. Time and memory requirements for direct-addressed instructions compared with other methods

computer can use this byte to decide when the end of the program occurs.

Direct addressing, though it can be used to address a byte anywhere in the memory, has the twin disadvantages of low speed and higher memory use. The speed of the instruction is slow because two more bytes have to be fetched following the instruction byte, and these bytes then have to be assembled into an address which is put onto the address lines to fetch the data. The demand for memory is also comparatively large because the two address bytes have to be stored immediately following the command byte. Figure 3.5 compares the time and memory requirements for some typical direct-addressed loads with other types of loads.

Register-indirect addressing

Indirect addressing means using one address (more precisely, two addresses) to pick up another one — its equivalent in everyday terms is going to the address of a tourist agency to find the address of a hotel. The method of indirect addressing that the Z-80 uses is called register-indirect addressing.

We have seen that the double registers BC, DE and HL can be loaded from an address in memory by instructions such as LD HL,(ADDR), and that this will result in the byte stored at address label ADDR being copied into the L register, and the byte stored at the next address, ADDR + 1, into the H register. This forms one half of a complete register-indirect load because if we then use the instruction LD A,(HL), the accumulator will be loaded from the address held in the HL register pair, and this address will be the one which was taken from the other two address locations, ADDR and ADDR + 1. Figure 3.6 shows an example which should help to make the process clearer.

Indirect addressing is a method that can permit some very powerful programming, because even if the main program is in ROM, the addresses which are used to hold the other address bytes (ADDR and

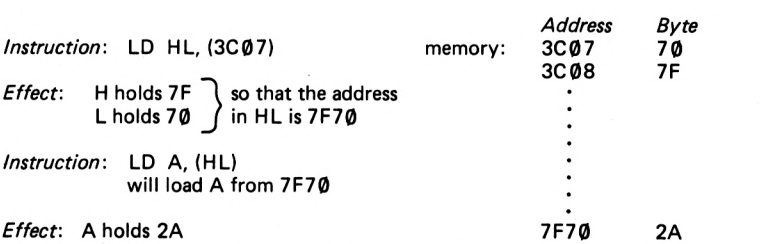


Fig. 3.6. The effect of LD HL,(ADDR) is to load two bytes from two addresses

ADDR + 1) can be in RAM, so that different bytes can be stored in them. You could imagine, for example, that ADDR and ADDR + 1 are originally loaded with the bytes that make up the address of a set of character bytes — the bytes which create the shapes of letters and numbers on the screen. By altering the address bytes in ADDR and ADDR + 1 we could make the Z-80 make use of a different set of character bytes, Greek letters, mathematical symbols, user-defined graphics or whatever we pleased.

In addition, when double registers such as HL hold an address number, that number can be incremented or decremented. Incrementing means adding 1, decrementing means subtracting 1, so that if we had HL loaded with the address Ø2ØØH, and we then executed the command LD A,(HL), the accumulator would be loaded with the byte stored at Ø2ØØH. If we then use the command INC HL and follow it with LD A,(HL) again, then the accumulator will be loaded this time from Ø2Ø1H because of the effect of INC HL. This form of loading is efficient because the commands LD A,(HL) and INC HL need only one byte each.

All of the double registers can be incremented or decremented in this way, so that the use of register-indirect loading is particularly advantageous when a set of values (called a table) has to be loaded, with one item after another. If the HL register pair is loaded with the address of the first byte of the table, then INC HL instructions can be used to 'shift the pointer', meaning make the address increment to a new value. This can be done in a loop, so that the INC HL command need appear only once.

Commands of this type are so useful that the Z-80 includes in its instruction set several 'block' commands, which will perform the actions of looping, incrementing and counting automatically once the registers have been set up. These commands are examined in detail in Chapter 5.

Another feature of register-indirect loading is that the byte stored in the address that is held in the register pair can also be incremented or decremented. Suppose, for example, that the HL pair holds the address 3CØ6H and that the byte stored at this address is 1BH. By executing INC (HL), the byte stored at 3CØ6H will be incremented to 1CH, but the address 3CØ6H is unchanged. This one command has performed the same action as the set of commands:

```
LD A,(3CØ6H)     ; put 3BH into A
INC A            ; increment to 3CH
LD (3CØ6H),A     ; put it back
```

(Note the convention that comments are added following a semicolon, which for assembly language is the equivalent of the REM in BASIC.) There is no command corresponding to INC (HL) for the BC or DE

register pairs, which is why the HL pair is the first choice for use in register-indirect addressing methods.

PC-relative addressing

Program-counter relative addressing (PC-relative) is a method which in many ways is a hangover from the early days of microprocessor design. A PC-relative addressed instruction is followed by a single byte, called the displacement, which is treated as a signed byte. To find the address that the instruction will use, this displacement byte is added to the address which is already present in the PC at the time of reading the displacement byte. This allows the instruction to obtain access to address numbers up to 127 steps forward from the current PC address or up to 128 steps back — any byte greater than 7FH will be taken as negative, causing the new address to be a lower number than the address of the instruction.

PC-relative addressing is used only for jump instructions of the Z-80 and only a comparatively few jump instructions make use of this type of addressing; the list is shown in Fig. 3.7. The formula for calculating the

Command	*Effect*
JR C, d	Jump if carry set
JR d	Jump
JR NC, d	Jump if carry not set
JR NZ, d	Jump if zero flag not set
JR Z, d	Jump if zero flag set
DJNZ, d	Jump if B-register content not zero

Fig. 3.7. A list of PC-relative addressed instructions

displacement byte for a jump of this type (a relative jump) is shown in Fig. 3.8. The convention is that the source address is the address of the JR instruction byte, and since the jump does not actually take place until the PC has been incremented to find the displacement, and since the PC will then increment again at the end of this fetch/execute cycle, the number that is jumped is always two more than we might expect, hence the subtraction of 2 in the formula. When an assembler is used, the label system will produce the displacement number automatically by statements such as:

```
JR LOOP
```

and the assembler will hang up with an error report if the jump uses a displacement which will not fit into a single signed byte.

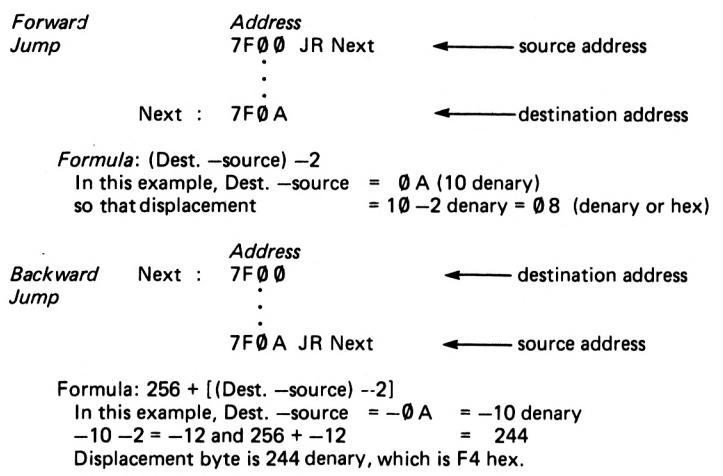


Fig. 3.8. Calculating the size of the displacement byte

Indexed addressing

Indexed addressing on the Z-80 is a form of register-indirect addressing in which a set of bytes that are needed in the program are stored in sequence starting at some address in RAM. This starting address is then stored in an index register using a command such as LD IX, F88ØH or LD IY, 7B1DH. Once the index address, often called the base address, is loaded, we can access any byte within the range of + 127 to –128 address numbers from this base address by using commands such as:

LD A,(IX + ØAH) or LD (IX–5),A

Two sets of index page addresses can be held, because both IX and IY registers are available for use. In addition to the use of the (IX + n) loading, the IX and IY registers can be incremented and decremented like the HL, BC and DE registers. Some care has to be taken when programs are written for a computer which makes extensive use of the IX and IY registers, because if the contents of these registers are changed, it will not be possible to use routines in the ROM. Programmers using the ZX Spectrum need to be particularly careful of this, and should check a disassembled listing of the Spectrum ROM before using these registers in their own programs.

Page zero addressing

Finally, the Z-80 uses a few commands which are zero-page addressed.

Zero page in this sense means that the higher byte of the address is ØØH, so that addresses such as ØØ1Ø, ØØ2Ø, etc, up to ØØFF, are all page zero addresses. The utility of this is that an address in this range consists of one byte only, so that access is fast, and only one operand byte is needed.

The Z-80, however, uses zero page addressing in a very limited way, for only one group of instructions, the RESTART (RST) instructions. These are the equivalent of CALL SUBROUTINE instructions, but are restricted to the list given in Chapter 5. Designers of computers put their most frequently used subroutines into these page zero addresses, so that they can be used by instructions of the type:

RST 1ØH

Since there are only eight bytes available at each RST address, only short routines can be placed there, but it is more normal to place a jump command, along with a different address, into three bytes, so that a longer routine can be carried out.

4

Assemblers

Programs should always be written in assembly language because its use makes programs easier to design and to follow, but programs can be entered into a computer in assembly language only if an assembler program is available. This is possible (with rare exceptions) only if the computer uses a Z-80 microprocessor, and some machines which do, notably the TRS-80 Model I and Model III, the Video Genie, the Sharp MZ-80K and the NASCOM, can make use of one of the best Z-80 assemblers, the ZEN assembler-editor. Other assemblers are available for other Z-80 based machines, most of them rather limited in comparison to ZEN. An alternative which is particularly useful for anyone who wants to learn about Z-80 programming but has a 6502-based computer such as an Apple, a Commodore or an Acorn/BBC, is to use a small Z-80 assembler machine, the MENTA unit. The MENTA unit permits entry of code in Z-80 assembly language mnemonics, generation and display of code whenever an instruction has been entered, facilities to run the program and display results, and a port system which will allow, given suitable software and hardware on the computer, transfer of code to the larger memory of another computer. Recent models of MENTA also permit disassembly of machine code, and printer output.

Though assemblers vary considerably, a reasonable working knowledge of any one type makes it relatively easy to learn the use of another type, so this chapter will be concerned with an outline of the use of these two useful systems, MENTA and ZEN. In the space of one chapter it is obviously impossible to cover both systems in detail, but the information should be enough for the reader to make an informed choice of assembler systems, knowing what should be possible.

TRS-80 ZEN

This is an assembler which is itself written in Z-80 machine-code, and which is remarkably compact in view of the facilities that it offers. When ZEN is loaded and running, a letter Z appears as a prompt on the screen

to indicate that a single-letter instruction is required. At the start of a new assembly language session, this command letter will be E, meaning ENTER assembly language. This, like all other ZEN commands, requires the use of the ENTER/RETURN key before the command is carried out, so that hasty key tapping does not have irreversible consequences.

The result of using E is that the Z-prompt moves up and a 1. appears, awaiting the first line of an assembly language program. This will usually be a title, and like all such comments which are not part of the assembly language itself, must be prefaced by a semicolon on each line. At each use of the ENTER/RETURN key a new line number will appear — the lines are in 1, 2, 3 . . . sequence. A program will start with the use of ORG followed by an address number to indicate the origin address for the program. If the code is to be assembled directly into the memory of the computer, the next line should read EXEC followed by the address at which the code is to be loaded in. This need not be the same as the address for which the code is written, but for testing purposes it is easier if the same address is used, particularly if the program is 'non-relocatable' meaning that if it is written for origin at one address it cannot be transferred to another address without modifying bytes of the code, particularly address bytes. Any program that contains memory references to addresses within the boundaries of the program itself (such as LD A,7F6CH or CALL 7FF1H in a program extending from 7F6ØH to 7FFFH) is non-relocatable. Unlike a program whose only memory references are JR displacements, a non-relocatable program will not run at another address unless all of these address references are changed. The only addresses which can be permitted are addresses in ROM, where the program cannot be shifted to. The ORG and EXEC commands of ZEN permit code to be tested at one address and re-assembled so that it will run at another set of address locations, provided that the memory references are in the form of label words which can be easily changed in the course of assembly.

Entry of the assembly language commands, which conform to Zilog conventions, continues until a full stop is entered, when ZEN returns to its waiting condition, with the Z-prompt showing. The assembly language program, known as the source code, can now be edited very much as a program written in BASIC could be, using the pointer and editing commands of ZEN. The pointer commands, used to locate and print lines in the assembly language program, are illustrated in Fig. 4.1. These can be used to check the program for missing, duplicated or incorrect lines and, once found, these can be dealt with by using the edit commands (Fig. 4.2). Z (Zap) removes the current line (the line which is printed when the P command is carried out) and will renumber all subsequent lines. E (Enter) will allow a new line or a set of new lines to be

T Move pointer to the top of the file (first line).

B Move pointer to the bottom of the file (last line).

D Move the pointer down the file. Using D by itself will move the pointer by one line. Using D with a number, like D7, will move the pointer down that number of lines. This does not produce any result on the screen.

U Move the pointer up the file. Used like D, so that we can have U12, to move the pointer up 12 (denary) lines. No effect on screen.

P Print the line pointed to. Will print the line at which the pointer has been set. If a number follows P, such as P5, then that number of lines, starting at the pointer position, will be printed. At the end of the command, the pointer points to the last line printed.

L Locate command. Will find the command in assembly language which is typed following L. For example, LSUB Ø2H will set the pointer to the first line in which SUB Ø2H occurs. If EOF is printed, then the command was not found, and the pointer is at the end of the file.

Fig. 4.1. The pointer commands of ZEN

entered, renumbering all of the lines that follow, until a full stop is entered. N (New) makes a new line, so that a line can be modified without changing the line number.

When the assembly language program is thought to be correct, it should be recorded. The method that is used for recording will depend on whether the cassette or the disc version of ZEN is being used, but in general the command W will write the assembly language source code to cassette or to disc. There is a validate (V) command that allows the recorded version to be compared with the program, which is still stored in the memory, to check for possible errors — though this is not usually necessary if a disc filing system is used.

Once the source code is safely recorded, the program can be assembled. The ZEN command is A, and this causes an option question to appear on the screen. This can be answered by V, which will direct all output to the screen alone. No code will be stored unless the EXEC pseudo-instruction was used in the assembly language, and this V option will normally be the first option to try, as we want at this stage to find out if the program will assemble correctly. If the program assembles correctly without any error messages, then the C command, which will create a recording of machine code, or the E command, which sends the code to a printer, can be used. The printed output is in numbered pages.

Z Zap—remove the current line. Can be followed by a number, so that Z7 will remove the 7 lines starting with the current line.

E Enter. Allows statements in assembly language to be entered until a full stop indicates the end-point. This can be used at the start of a session, or to insert code in an existing program. Line numbering is automatic in either case.

N New. Allows the current line to be edited. Using the back arrow will space back from the end of the command, allowing changes to be made. The end of editing is signalled by using ENTER/RETURN.

Fig. 4.2. The edit commands of ZEN

DOUBLE SYMBOL	You have tried to use the same label name for two different purposes
UNDEF	You have used a label name without assigning a meaning to it
RSVD	You have used a reserved word like ORG or END as a label name
SYMBOL	You have omitted a label name
FULL	Out of memory
EOF	End of file—you forgot to put an END statement
ORG	You forgot to put an ORG statement in
HUH?	The line doesn't make sense, impossible command, misspelling, etc.
OPND	Something wrong with an operand

Fig. 4.3. The ZEN error messages

If there are errors in the assembly language which make assembly impossible, such as incorrect syntax or impossible displacements for JR commands, then assembly will halt and an error message (Fig. 4.3) will be displayed. The error can then be corrected using the editing commands of ZEN, and assembly tried again. The later versions of ZEN print the symbols (label names), sorted into alphabetical order, that are used in the program when the appropriate command is used. This allows you to find the label names in long programs very easily, and a similar command XREF will cause a cross-referenced list of label names to be printed with a list of lines in which these label names appear.

Once assembly has been achieved successfully with output to the screen, the program can be re-assembled with the machine code sent to cassette or disc. If the EXEC command has been used, or if this is edited into the source code before final assembly, then the machine code can be placed directly into the memory of the machine, and ZEN provides for clearing an area of memory for this purpose. The command H (Howbig?) can be used at this stage to check the size of the source-code program (in hex), and K (Kill) can be used after the source code has been recorded, to clear the memory ready for another assembly language program, which can be entered from the keyboard or from tape or disc.

If code has been placed directly into memory, it can be checked and run by making use of the monitor commands of ZEN. These are illustrated in Fig. 4.4. The G (Goto) command allows the code that has been loaded into memory to be run — the ultimate test of a machine-

G	Goto the address following the G command and execute the program
H	Howbig. Returns with the number of bytes of source code
F	Fill a block of memory with a data constant
M	Modify memory, starting with address given. Will print the memory address and contents, and wait for you to type a new byte if this is to be changed. If no change is needed, press ENTER/RETURN
X	Examine the Z-80 register contents
Q	Query—examine a block of memory, starting at the address following the Q command

Fig. 4.4. The monitor commands of ZEN

code program. If all is well, but the program does not do quite what was expected, then other monitor commands can be used to examine and possibly modify the code. Generally, however, if a machine-code program fails, it fails in a big way, and the machine will have to be reset. ZEN itself is usually unaffected by a reset, so that the source code can be reloaded from tape or disc, using the R(Read) command, and any necessary alterations made.

The advantages of ZEN as compared to earlier designs are:

1. Very compact code — ZEN takes up surprisingly little RAM space, even in a 16K computer;
2. Simplicity in use — ZEN is easy to learn and to use;
3. Robustness — ZEN will generally survive machine-reset operations;
4. Flexibility — ZEN is a combined assembler, editor and monitor which can be used to develop machine-code programs without the need to load any other programs.

Potentially, ZEN could be used with any Z-80 based computer, but at the time of writing is available only for the types named earlier. It is very much to be hoped that this excellent program will be made available for other Z-80 based machines, so that owners of machines other than those mentioned can discover for themselves the advantages of this excellent system.

MENTA

MENTA is a completely different approach to the subject of Z-80 assembly language programming, since it consists of a complete machine with the assembler in ROM. The hardware consists of a Z-80 microprocessor, with 8255 programmable port, 1K of RAM, ROM, and a character display generator. The port connections are taken to a 26-pin fixed plug to which a socket can be connected to allow the MENTA unit to exchange signals with other equipment. There are also sockets for connections to a TV receiver (used for display), to a power supply, and to a cassette recorder.

The MENTA keyboard (Fig. 4.5) is a membrane type with the keypads labelled with the Z-80 mnemonics, covering all but the least-used Z-80 instructions. By making use of these keys a program can be typed directly in assembly language, using the one-key-per-instruction method pioneered by the ZX-80; though in the case of assembly language, the opcode needs one key and each part of the operand will require separate keys to be used. Even the normally awkward business of calculating jump addresses can be carried out automatically. The shortcomings of the membrane keyboard are partly overcome by arranging for a short beep from a built-in loudspeaker to sound when a

Fig. 4.5. The appearance of MENTA. The photo shows a very early pre-production model, hence some of the labels on the keyboard!

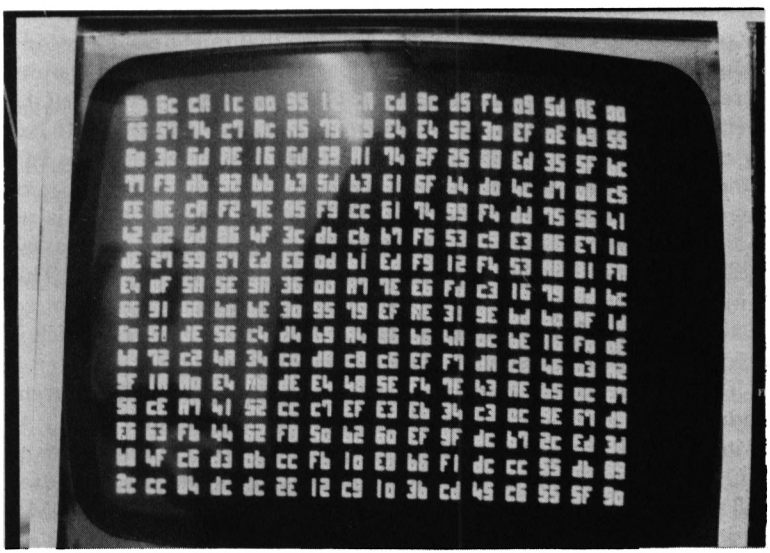


Fig. 4.6. The appearance of one page of the MENTA display

key has been successfully pressed. This note becomes a longer squawk if the entry is an impossible command.

The machine code generated by the assembler is displayed on the screen of the TV receiver (Fig. 4.6) plugged into MENTA. The screen shows the contents of 256 bytes of RAM at a time, starting normally with the first available address, which is F8ØØH. A program written using MENTA would normally be located with its first byte at this address, and the screen would display addresses ranging from F8ØØH to F8FFH. Each block of 256 bytes is known as a page, and pages can be selected to be displayed on the screen by pressing the PAGE key and following it with the number Ø, 1, 2 or 3 — the LED display next to the keyboard will remind you which page is being displayed.

The program stored in the RAM of MENTA can be displayed, edited and altered as needed, with insertions and deletions of code, and then run. A program can also be recorded on cassette — the system that is used actually records the whole of the RAM contents — and replayed for future use. Note however, that it is the machine code which is displayed and recorded, not the source code. As usual, it makes good sense to record a program before trying to run it.

Procedure

With MENTA connected up, switching on produces a set of meaningless garbage bytes on the TV screen. These can be reset to zero by pressing the CLEAR key twice. Any key that might call for second thoughts requires two depressions to operate — this is a safety feature, necessary because MENTA has no ENTER/RETURN key. When the memory is zeroed in this way, the display will show 16 rows of 16 hex bytes, all zero, with a flashing cursor over the first byte at the top left-hand corner. The section of memory that is displayed is page zero, memory addresses F8ØØH to F8FFH inclusive, though there is no indication of the memory addresses on the screen. As a reminder, the LED display on the right of the keyboard will show the figure Ø. This initialisation procedure does not zero all of the RAM, because there is a section of memory at the end of page 3, FBC7H to FBFFH, which is used by MENTA's operating system (as part of its stack).

The flashing cursor displayed on the screen indicates the 'current address', the location at which pressing a key will deposit a byte. At switch-on and when cleared, the cursor is in the position referred to as HOME, located on the screen at the top left-hand corner. The cursor will move along as bytes are entered, and can be returned to the HOME position at any time by pressing the HOME key. This action does *not* change the page number, however, it simply returns the cursor to the

HOME position on that page. The keys that are marked UP, DOWN, LEFT and RIGHT will move the cursor in the directions indicated; the keys have autorepeat and can range over the whole memory. Holding down RIGHT will, for example, cause the cursor to move to the right-hand edge of a line, then reappear, still moving right, on the next line down. If this is done with the cursor on the bottom line of a page, the cursor will then appear at the top of the next page.

If, at any location of the cursor, the STORE key is pressed, the address of that position will be memorised and can be inserted into any memory command as an address or a displacement. In addition, pressing RECALL at any subsequent time will return the cursor to the stored address position, providing that another address has not been stored between the time of the original STORE operation and the use of RECALL. This position storage system is particularly useful for inserting JUMP addresses.

Program entry

MENTA can be used to enter programs written in hex by simply treating the keys as hex entry keys, using the markings on the keypads themselves. The cursor will move one step on for each two hex digits, corresponding to one byte, and the program should end with the byte FFH which will be read as an instruction to cause a return to the MENTA operating system. The cursor permits insertion or deletion of code at any point.

This facility, however, is less important than that for entering assembly language. With the cursor placed at the starting address for the program, which need not be the HOME position, the ASMBL key is pressed. This will cause the top bar of the LED display to light. This is a reminder that the next key-stroke will generate the code for one of the assembly-language mnemonics that are shown above each keypad. When such a key has been pressed, the LED display will light the bottom bar and extinguish the top bar as a reminder that the next byte must be generated by selecting a register or flag reference situated underneath a keypad position. Any data, bytes or addresses can also be entered following this, and the LED will return to displaying the top bar when all of the data that is required for one complete instruction, corresponding to one line of assembly language, has been entered. The screen will then indicate the code with the instruction byte highlighted, a useful guide to the position of the machine code on the screen. If an impossible combination of opcode and operand is attempted, the loudspeaker will render a protest, a sound which is quite different from the short beep which greets each successful key-press.

A few Z-80 instructions which are seldom used are not catered for in this way and have to be entered directly in hex. This is done by pressing ASMBL again to leave the assembler mode, and then entering the hex. This also has to be done at the end of a program so that the terminator byte FF can be entered.

Jump addresses are entered by leaving assembly mode, moving the cursor to the address which is the destination for the jump and pressing STORE. The cursor is then moved back to the start of the jump instruction. When ASMBL is pressed again, the jump instruction can be entered, but this time pressing the RECALL key instead of typing an address or displacement operand. This inserts the address for a JP and the displacement for a JR automatically, and is particularly useful for JR jumps.

Once the program has been entered and checked, it should be recorded. MENTA uses one lead between itself and a recorder, and this will have to be connected to the MIC input of the recorder. There is no motor control — the recorder is started, with RECORD and PLAY keys pressed, and the TRANSMIT key of MENTA pressed. The screen will go blank for the few seconds that are needed to record the entire contents of the RAM and will return when the recording is complete, when the recorder can be switched off.

Running and checking

A program that has been entered and recorded can be run by moving the cursor to the first byte of the program and then pressing the RUN key twice. If the program contains an indefinite loop (accidentally or deliberately), the control can be recovered by pressing the INT key (in serious cases, the BREAK key). If the program proceeds in an orderly fashion, the cursor will reach the last byte of the program on the screen and the operating system will be restored. The screen is always blank when a program is running. The effects of the program can then be checked by examining the contents of memory locations affected by the program. This is done by reading the byte on the appropriate page — since all bytes on one page are visible at one time, several memory locations can be read at a glance.

The state of the Z-80 can be inspected by pressing the REG key, which places the display to page 3 with the cursor over the point where the contents of the flag (F) register are stored on the stack. On the right of this position are stored the contents respectively of the A,C,B,E,D,L and H registers.

The ability to inspect memory and register contents like this is particularly valuable when single-stepping is used. When the cursor is

placed over the start of any instruction byte, pressing the STEP key will cause that single instruction to be executed. This may cause the cursor to move several steps on, because an instruction may require one or more bytes of data to be read before it can be executed. After such a single step, the contents of memory and of registers can be inspected to check that the instruction has had the desired effect. If RECALL is now pressed, the cursor will return to the instruction byte which is next in execution order, so that another STEP will carry out this one. The use of this technique allows you to single-step through an entire program, inspecting memory and registers at each step.

The MENTA system is particularly useful for the beginner or for anyone developing short sections of machine code for use in control programs. It is recommended to any learner whose computer cannot accept a Z-80 assembler either because a different chip is used or because no suitable assembler is available for that computer. At the learning stage, MENTA can be used entirely on its own, but as your confidence in machine-code programming develops, you can make use of MENTA to write code for other systems, and then explore the possibilities of using the PORT inputs and outputs to transfer code between MENTA and other Z-80 based machines. One important point to remember, however, is that address references may have to be changed, because it is unlikely that the computer to which code is being transferred from MENTA will use the same address range as MENTA, F8ØØH onwards. However, if the program is written in entirely position-independent code, using no full addresses, this will not be necessary.

5

Instructions

In this chapter we shall look at the various types of instructions that make up the instruction set of the Z-80. These have been broadly classed as: transfers; increment and decrement; test, jump and call; arithmetic and logic; shift and rotate; input/output; and miscellaneous. There are instructions which can't be classed rigidly into one group, but which for the sake of convenience have been placed in the group that seems most appropriate.

Transfer instructions

Register-to-register transfer

The contents of any single-byte working register, in particular A,B,C,D,E,H and L can be copied to any other of these registers, using commands of the form:

LD A,H or LD C,D

which consist of one byte only. These register-to-register transfers are economical in program space because they use a single byte and are also fast to execute (generally four clock cycles) so that this method of loading a register is extensively used in Z-80 programs. It is easier, for example, to store a byte temporarily in a spare register than to keep it at some memory address because the action LD A,C, for example, can be carried out much more rapidly and with less use of memory than one such as LD A, (ADDR). This is because no memory address signals have to be sent out on the address lines when these register-to-register transfers are carried out.

There are two register-to-register transfers which are slower than these examples. The operations involving the I and R registers are carried out indirectly by the Z-80, and are very much slower than transfers between the 'working' registers. Fortunately, these transfers are very rarely needed and, when they are used, the time needed is of little importance.

There are a few transfers which can be caried out between the double registers. The stack pointer can be loaded from the HL pair using LD SP, HL and also from IX or IY, but the only other double-register transfers are exchanges in which the contents of two sets of registers are swapped over. Of these, EX DE, HL is the most useful, though the alternate register instruction EX AF, AF', which brings the alternate AF set into use in place of the normal AF set in all instructions involving the accumulator and the status register, is sometimes used. Another instruction of this type is EXX, which brings in the alternates for BC, DE and HL.

Register-memory transfer

All of the common single-byte registers, A,B,C,D,E,H and L can be immediate-loaded from memory using an instruction of the form LD D,2FH, which consists of two bytes. The time needed is seven clock cyles. The same set of seven registers can be loaded indirectly from the address contained in the HL register pair, or by indexing using the IX and IY index registers. Indirect loading from the HL address needs seven clock cycles, indexed addressing is very much slower, requiring 19 clock cycles.

The accumulator, but not the other single-byte registers, can be loaded directly, needing three bytes and 13 clock cycles, or from the contents of BC or DE. The LD A, (BC) and LD A,(DE) instructions are both single-byte and need seven clock cycles for completion.

Instruction	*Code*	*Time*
LD A,(ADDR)	3Aaddr	13
LD A,(BC)	ØA	7
LD A,(DE)	1A	7
LD A,(HL)	7E	7
LD A,(IX+d)	DD7Ed	19
LD A,(IY+d)	FD7Ed	19
LD A,A	7F	4
LD A,B	78	4
LD A,C	79	4
LD A,D	7A	4
LD A,E	7B	4
LD A,H	7C	4
LD A,L	7D	4
LD A,I	ED57	9
LD A,R	ED5F	9
LD A,N	3En	7

Fig. 5.1. Instructions for loading the accumulator, showing codes and times. In the table, ADDR means a two-byte address, d is a displacement byte, and n is a data byte

All of these loads, summarised in Fig. 5.1 with the exception of the immediate loads, can be carried out in the reverse order, register to memory, using the same number of bytes or code and needing the same number of clock cycles. Just to take an example, LD (HL),B is a single-byte instruction which needs seven clock cycles, and LD (ADDR),A is a three-byte instruction which needs 13 clock cycles.

Increment and Decrement

Any of the single-byte registers A,B,C,D,E,H or L can be incremented or decremented by using instructions such as INC B, DEC C and so on. The instructions consist of one byte each, and the time is four clock cycles for each action. Increments and decrements of these single-byte registers will affect the S and Z flags (also H and P/V), but *not* the carry flag.

A byte stored in the memory can also be incremented or decremented if its address is held in a register pair. The command INC (HL) will, for example, increment the byte that is stored at the address held in the HL register pair. This instruction also consists of a single byte, but because the memory reference is indirect, the time needed is 11 clock cycles. The main flags are the S and Z flags (H and P/V also); the carry flag is not affected. Bytes contained in an address accessed through use of the index registers can also be incremented or decremented, but the instructions require three bytes and a time of 23 clock cycles. The same flags are affected.

Register pairs can also be incremented or decremented using instructions such as INC HL or DEC BC. This form of instruction causes the number held in the register pair to be incremented or decremented, and you must be careful to distinguish this from the indirect single-byte INC or DEC such as INC(HL) or DEC (BC). When register pairs are incremented or decremented, *no* flags are affected so that if the result of incrementing or decrementing a register pair must be tested, some other method of setting flags will have to be used. One favourite method is:

```
DEC BC              ;decrement
LD A,B              ; byte from B to A
OR C                ; OR A with byte from C
JR NZ,Back          ; jump step if result not zero
. . . . . .         ; other instructions
```

Since the DEC BC action does not affect the flags, the accumulator is loaded from B and ORed with C. Only if both of these bytes are zero will this OR action set the zero flag, so that this will provide the test that is needed. The inability to set flags from the INC or DEC operations on the

double registers is a shortcoming of the Z-80 which was inherited from its ancestor, the 8080.

The INC and DEC instructions applied to the registers BC, DE, HL or SP require one byte and take a time of six clock cycles. The index registers IX and IY can also be incremented and decremented, but need two instruction bytes, and the time is ten clock cyles. As for the other INC and DEC operations on double registers, no flags are affected.

Test, jump, and call

The Z-80 equivalent of BASIC IF . . . THEN is the test, the equivalent of GOTO is the Jump instruction, and the equivalent of GOSUB is the Call instruction. These actions are grouped together because they all cause the sequence of execution of a program to depart from the simple address sequence that is achieved by incrementing the program counter. Testing is carried out by checking the flags of the F register, and the test conditions are always associated with either the jump or the call commands. None of these commands affects flags, though they may make use of flags in determining whether or not the jump or call takes place.

A jump command causes the next instruction to be taken from an address which is *not* the address immediately following the jump bytes. Once a jump has been executed, the normal sequence of incrementing the PC will resume at the new address (Fig. 5.2). A CALL, by contrast, will cause a new address to be placed in the PC and a new piece of program to be executed, but only until a RET instruction byte is encountered. At the RET instruction, equivalent to the RETURN part of a GOSUB in BASIC, the program resumes with the instruction that follows the CALL instruction, just as if the CALL instruction had not existed. The similarities between Jump and GOTO and between CALL

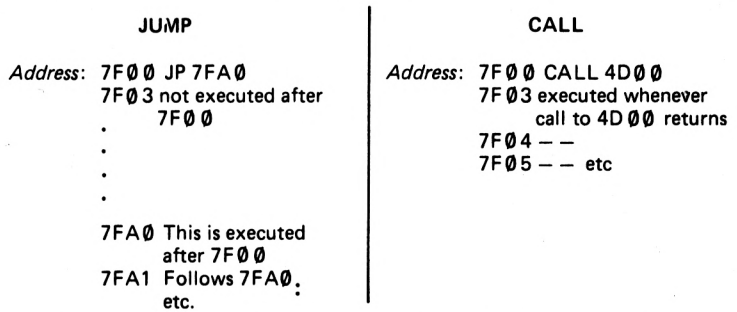


Fig. 5.2. The differences between JUMP and CALL. CALL always implies a return to the instruction that follows CALL, JUMP does not

and GOSUB are very marked — there is a closer correspondence between BASIC and assembly language here than in any other part of assembly language.

The unconditional jump that uses direct addressing is JP. The instruction needs one byte followed by an address, three bytes in all, and the time taken is ten clock cycles. No flags are affected. Its PC-relative counterpart, JR, is followed by a single displacement byte, so that two bytes in all are needed, but the time is greater, 12 clock cycles, because the microprocessor has to calculate the address for the jump using the PC address and the displacement byte. The main advantage of using relative jumps is that their use makes the program position-independent, meaning that the code can be located and used at any address without having to be re-assembled. This assumes, of course, that no other instructions in the program use full addresses from within the program address range.

Condition	*Flag*	*Meaning*
NZ	Z	jump if not zero
Z	Z	jump if zero
NC	C	jump if carry not set
C	C	jump if carry set
PO	P/V	jump if parity odd
PE	P/V	jump if parity even
P	S	jump if positive
M	S	jump if negative

Fig. 5.3. The conditions that can be used to control JP jumps

The direct-addressed jump, JP, can be followed by a condition operand from the list shown in Fig. 5.3. There are eight possible conditions, of which the most commonly used are the ones that involve the Z or C flags; less often the S flag. Like the unconditional jump, these instructions need a total of three bytes and the time taken is ten clock cycles. The PC-relative JR with a condition operand has a smaller number of choices of conditions, C and Z only. Two bytes are used, one being the displacement, and the time taken depends on whether the condition that is being tested is true or false. If the condition (C, NC, Z or NZ) is true 12 clock cycles are needed, but if the condition is false only seven clock cycles are needed, because the displacement has not been added. Some care must be taken when calculating the time taken by any set of instructions that makes use of these PC-relative jumps.

There are three indirectly addressed jumps, JP(HL), JP(IX) and JP(IY). Each of them causes a jump to an address held in the respective double register and as usual, the (HL) jump is the most efficient, consisting of a single byte with only four clock cycles of time needed. The

index register addressed jumps need two instruction bytes and take eight clock cycles.

The CALL instructions exist only as direct-addressed instructions, though a relative-indexed call would be extremely useful and could be implemented using RST with no change to the Z-80. An unconditional call requires a total of three bytes and a time of 17 clock cycles. In the course of the instruction the address that exists in the program counter just before the call is executed is stored in the stack memory so that a return can be made, and it is up to the programmer to ensure that the stack pointer has been loaded with a suitable address in RAM before this command is encountered. It is also a matter for the programmer to ensure that any stack operations that are carried out during the course of the subroutine call do not affect the return address that has been placed in the stack memory. Programs can be written which deliberately change this address under some conditions, but such program methods are hard to follow and are undesirable unless they produce results that cannot otherwise be achieved.

The conditional calls, such as CALL NZ, 2A46H, make use of the same set of conditions (see Fig. 5.3) as the direct-addressed jumps. Three bytes are needed, and the time taken depends on whether or not the call is actually made. If the condition is true and the call is made, 17 clock cycles are needed, but if the condition is false and the call is not made, then only ten clock cycles are needed.

The RET command, which is used to return from a subroutine call, consists of one single byte and needs a time of ten clock cycles. The action of the RET command is to take the original address bytes from the stack memory and restore them to the program counter, so that the success of this action depends on these bytes being at the correct position in the stack when the RET command is executed. More precisely, it depends on the stack pointer holding the correct address. I have stressed this point, because one of the most fruitful sources of baffling program failure is incorrect use of the stack which leads to the incorrect pair of bytes being used as a return address.

The RET instruction can also be made conditional, using the same set of conditions as the JP command. The RET command is still a single-byte one, but the time needed is 11 clock cycles if the condition is true and five if the condition is false. Other return commands that concern interrupts will be dealt with later.

The restart (RST) instructions of the Z-80 are a specialised form of call instruction that is shorter and quicker because it uses page zero addressing. The instruction byte exists in eight forms which differ in respect of the address in page zero to which each call is made. These are written in assembly langugage as RST ØH, RST 8H, RST 1ØH etc., which makes them look as if a displacement byte were being used, but in

Mnemonic	*Address*	*Code*
RST 00	00H	C7
RST 08	08H	CF
RST 10	10H	D7
RST 18	18H	DF
RST 20	20H	E7
RST 28	28H	EF
RST 30	30H	F7
RST 38	38H	FF

Time = 11 cycles

Fig. 5.4. The RST commands, using page-zero addressing

fact each instruction consists of one byte only, whose forms are detailed in Fig. 5.4. Each RST instruction causes a call to an address 00XX, where XX is the byte specified by the form of the instruction. The address which exists in the PC at the time when the RST instruction is encountered is placed in the stack memory, and the programmer has to ensure that a suitable piece of program exists at each RST start address that will be used. Since the addresses permit only eight bytes of program each, these routines are necessarily short, and in some cases may consist of no more than JP instructions. There must be a RET command at the end of each routine which is called by RST so as to make it possible to resume normal action.

Arithmetic and logic

Next to the load and transfer group, this is the largest group of Z-80 instructions, as it is for other microprocessors, and we shall divide the group into the 8-bit group and the 16-bit group.

Eight-bit arithmetic instructions consist of ADD and SUBTRACT only, and these apply firmly to the accumulator. All addition and subtraction operations concern two bytes one of which must be present in the accumulator (though this byte could be zero) before the instruction is used. The other byte can be taken from any of the single-byte registers, or from memory, using the whole range of addressing methods with the exception of relative and zero-page. The result of the operation will then be placed in the accumulator, and if there is an overflow, so that the result needs nine bits rather than eight, then the carry bit will be set in the flag register. The register-to-register arithmetic operations take four clock cycles, the register-indirect addressed actions need seven clock cycles, as do the immediate addressed actions, and the indexed instructions take 19 clock cycles.

Suppose, for example, that the accumulator stores 28H and the

command ADD A,C is executed. If the contents of the C-register are the byte 17H, then the result, which is 3FH is stored in the accumulator, the C-register still contains 17H, and the carry flag is reset, zero. The S and Z flags are also affected by the arithmetic operations (both will be zero in the example), as also are the H and P/V flags which we shall deal with shortly.

Suppose now, to take another example, that the accumulator had been loaded with 2AH before the ADD command, and the ADD A,H was executed with the H register containing 7CH. The result of this addition is 11EH, and the effect will be that the accumulator stored the byte 1EH, with the carry flag set to indicate that the most significant bit of the number, the ninth bit, is 1.

We can now examine in more detail the actions of the other flags. The H-flag is set if there is a carry from bit 3 of the accumulator. This facility

(a)
  01010110 denary 86
+ 00011011 denary 27

  01110001 denary 113 There has been no carry into the msb, so that whether we regard the numbers as signed or unsigned, the result of the arithmetic is correct.

(b)
  11001011 denary 215 (unsigned) or −53 signed
+ 11010111 denary 203 (unsigned) or −41 signed

= 10100010 + carry bit. Unsigned, this represents 215 + 203 = 418, counting the carry bit as representing 256. Signed, and ignoring the carry, the result is −94, which is also correct if the bytes are taken as being signed.

(c)
  10010110 denary 150 (unsigned) or −106 signed
+ 10001011 denary 139 (unsigned) or −117 signed

= 00100001 + carry set. If taken as unsigned, and adding value of carry bit (= 256) then result is 289, which is correct. If taken as signed, the result is +33, which is incorrect. If the answer is taken as the twos complement of the correct answer, then by inverting the twos complement procedure (i.e. subtract one and invert each bit), the answer is found to be −223, which is correct, though an unorthodox way of using the byte!

(d)
  01001101 denary 77 signed or unsigned
+ 01000111 denary 77 signed or unsigned

= 10010100 no carry. This is correct (148) if the bytes are regarded as unsigned, but if the bytes are regarded as signed, then this is −108, which is incorrect.

In each case if there has been a carry from the msb or from the bit next to it, but not from both, then the answer to signed arithmetic will be incorrect and must be corrected if signed arithmetic is being used.

Fig. 5.5. The use of the overflow flag for arithmetic. This flag is set if there has been the same sign of carry from both bit 6 and bit 7 in the accumulator

is used only when the bytes that we are dealing with are regarded not as binary-code bytes but as pairs of BCD coded numbers. The P/V flag is set if an 'overflow' occurs, and this type of overflow must be distinguished from a carry. The term 'overflow' refers to the use of signed arithmetic, using seven bits to represent the size of a number and the eighth, the msb, to represent sign. A carry from the seventh to the eighth bit could then cause the resulting byte to have the wrong sign, and the wrong value, and this state is signalled by the P/V flag (Fig. 5.5). Technically, this flag is set by XORing the carry out of the seventh and eighth places of the accumulator during an arithmetic operation. The P/V flag causes no automatic action, so that if the programmer wants to write a program that uses signed bytes, it is up to him/her to make use of this flag (as a JP condition, for example, or as a CALL condition).

The carry bit is treated quite differently, because it can be used automatically. Take, for example, the two Z-80 addition commands ADD and ADC. ADD means 'add to the byte in the accumulator', and its use can cause the carry flag to be set. It does not, however, take any notice of the state of the carry flag before the operation is carried out. By contrast, ADC will add to the byte in the accumulator the byte in another register or memory, plus the carry byte as well. For example, if A contains 2FH and C contains 13H and the carry bit is set, then ADC A,C will give the result 43H in the accumulator, with the carry bit reset, because there is no carry out from the addition.

This automatic use of the carry bit with certain instructions such as ADC means that numbers of as many bytes as you want to use can be added, low bytes first using ADD, then higher bytes in turn using ADC, providing that the results are transferred to memory after each addition. The process of addition, starting with low bytes and progressing to higher bytes is aided by the method that the Z-80 uses to store multi-byte numbers, with the lowest byte stored at the lowest of the address numbers.

The addressing methods that can be used with ADD and ADC are illustrated in Fig. 5.6; remember however that the results are always put into the accumulator. As usual, the register transfers are the quickest,

Instruction	*Time in cycles*
ADD A,(HL)	7
ADD A,(IX+D)	19
ADD A,(IY+D)	19
ADD A,A	4
ADD A,B	4
ADD A,C	4
ADD A,D	4
ADD A,E	4
ADD A,H	4
ADD A,L	4
ADD A,DA	7

Fig. 5.6. ADD addressing methods, with execution times. The same set of addressing methods and times also applies to ADC

needing four clock cycles, the indexed additions are the longest in terms of bytes and time, needing 19 clock cycles.

The corresponding subtraction commands are SUB and SBC. Using SUB will subtract from the byte in the accumulator the byte at the register or address specified in the instruction. The result will be stored in the accumulator, and the carry flag will be set if there has been a 'borrow', meaning that the byte in the accumulator before the instruction was executed was smaller than the byte being subtracted. The S,Z,H and P/V flags can be affected by these subtraction operations, and the N flag is always set to indicate that a subtraction is being carried out. The corresponding SBC command carries out the same process as SUB but will also subtract the carry bit value (0 or 1), from the byte in the accumulator. Once again, we would use SUB for the lowest order bytes in a multi-byte subtraction, with SBC used for all the higher order bytes. The range of addressing methods and the times follow the same pattern as for ADD and ADC.

A command which also fits into this group is CP. CP carries out the same action as SUB, but no result is placed in the accumulator, so that the byte in the accumulator after executing a CP command will be the same as it was before the command was executed. Using CP, however, will affect flags, so that this command lets us set flags in the same way as SUB but with no other result. If, for example, the accumulator is loaded with 3BH at some stage, and then the command

CP 3BH

is executed, then the Z flag will be set, but the number in the accumulator will still be 3BH. Using CP 3CH would cause the C flag to be set, along with the S flag. Using CP 3AH would not cause any flags to be set. The list of addressing methods for CP is as for ADD, Fig.5.6. According to the addressing method used, the command may need one to three bytes, and times of four cycles (register-to-register) to 19 cycles (indexed).

The double-byte arithmetic instructions are not so numerous in variety as the single-byte types. Both types of addition (ADD,ADC) can be carried out using HL as an accumulator, with any of the other register pairs BC, DE, HL itself, or SP; the result of addition is stored back into HL. The instructions use a single byte, and the time needed is 11 clock cycles. The C flag will be set if there is a carry out of the msb of the H register, reset otherwise, and the S and Z flags are *not* affected, though the H flag is set when there is a carry out of the bit number 11 (numbering the lsb as Ø). The corresponding ADC HL,XX instruction will, as you might expect, carry out the same action but with the automatic addition of the carry bit. The other change is that ADC will require 15 clock cycles.

The only 16-bit subtraction is SBC HL,XX where XX is the same choice of double registers as shown above for ADD and ADC. The action of SBC is to subtract from the HL register the contents of the other register pair, along with the carry bit, so that if the status of the cary bit is unknown and no carry is to be subtracted, the SBC command must be preceded by some command which will reset the carry flag but which will not otherwise affect the program. A good choice is AND A, which will leave the contents of the accumulator unaffected but which will reset the carry flag; OR A will have the same effect.

Addition without a carry, using ADD, can be carried out using the index registers IX and IY, and any other double registers chosen from a group illustrated in Fig. 5.7, though IX cannot be added to IY. The

```
ADD IX,RR     RR can be: BC, DE, IX, SP
Result stored in IX
Time : 15 clock cycles
Memory use : 2 bytes
Flags : N flag reset, H and C flags affected, others unaffected.

ADD IY,RR     RR can be: BC, DE, IY, SP
Result stored in IY
Otherwise as above
```

Fig. 5.7. The addition instructions affecting the index registers

Instruction	*Time*
AND(HL)	7
AND(IX + N)	19
AND(IY + N)	19
AND A	4
AND B	4
AND C	4
AND D	4
AND E	4
AND H	4
AND L	4
AND n	7

Fig. 5.8. Addressing methods and times for the AND instruction. The same methods and times apply to the other logic commands

Bit comparison:

Bit 1	Bit 2	Result
0	0	0
0	1	0
1	0	0
1	1	1

Byte comparison:

```
1st byte :  1 0 0 1 1 1 0 0     Hex 9C
2nd byte :  1 1 1 0 0 1 1 1     Hex E7
Result :    1 0 0 0 0 1 0 0     Hex 84
```

Fig. 5.9. The action of the AND command. This compares corresponding individual bits in two bytes, and does not affect the carry flag

result of the addition is placed in IX, and the main flag that is affected is the carry flag. The instruction takes two bytes and 15 clock cycles.

The logic group is a smaller set of instructions which once again require one of the bytes to be placed in the accumulator before the instruction is executed. The AND instruction can apply to bytes addressed as shown in Fig. 5.8 and the result is placed in the accumulator. The operation consists of comparing bits (a bitwise operation) in the two bytes. If two bits in corresponding position are both

```
   Byte  10110110      Hex B6
AND ØF   00001111
Result   00000110      Hex Ø6
```

Fig. 5.10. Using an AND instruction to 'mask off' part of a byte

Bit comparison:

Bit 1	Bit 2	Result
0	0	0
0	1	1
1	0	1
1	1	1

Result is 1 if either bit (or both) is 1.

Byte comparison:

```
          1st byte    10110100    Hex B4
          2nd byte    01100001    Hex 61
OR        Result      11110101    Hex F5
```

Fig. 5.11. The OR action

Bit comparison.

Bit 1	Bit 2	Result
0	0	0
0	1	1
1	0	1
1	1	0

Result is 1 if bits are dissimilar, 0 if identical

Byte comparison:

```
          1st byte    01101101    Hex 6D
          2nd byte    11110000    Hex FØ
                      10011101    Hex 9D
```

Clearing a register:

```
          A content   : 01101010
          XOR with    . 01101010
                      ----------
          Result      00000000
```

Fig. 5.12. The XOR action. Note how XOR A can be used to clear the accumulator; this is much faster than LD A,00, for example

1, then a 1 bit is placed in the corresponding place of the acumulator as a result (Fig. 5.9). The main flags affected are S and Z, the H flag is set, and the N and C flags are reset. The AND function is often used as a 'mask' to select part of a byte; for example, AND ØFH will force the top four bits of a byte to be zero (Fig. 5.10).

The OR operation, illustrated in Fig. 5.11, operates in a similar bitwise way to AND, and uses the same addressing methods. Both of these instructions take one to three bytes (1 for OR C, 3 for OR(IX + N)), and the times range from four cycles for a register-to-register transfer to 19 for an indexed address. The third command of this group, XOR, uses the same addressing methods, but the logic of the command is as shown in Fig. 5.12. This leads to the use of XOR as a quick way of clearing the accumulator, because XORing a byte with itself gives a zero result. The command XOR A is faster and less memory-consuming than LD A,ØØH.

Shift and rotate instructions

The operations of shifting and rotating consist of altering the positions of every bit of a byte contained in a register or in memory. A single shift or rotate command will shift each bit by one place. The relative positions of bits are preserved, except at the ends of registers (msb or lsb), but the hex value of the byte considered as a number will be changed. Shifting implies that bits are lost from one end of the register, with 0 shifted in at the other end each time a bit is shifted out; rotation causes the ends of the register to be connected so that bits are not lost. A bit leaving the lsb position of a register will move to the msb position, for instance. Most of the rotation and shifting commands of the Z-80 also involve the carry flag, some treating it simply as a flag, others using it as a storage space for a ninth bit.

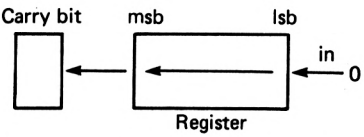
The SLA and SRL commands are illustrated in Fig. 5.13(a). These cause the bits of a byte to be shifted left or right respectively, with one bit shifted to the carry, and a zero placed into the register at the other end. The operand for the instruction can be any of the single registers, the (HL) byte, or a byte obtained by indexed addressing; the times and number of bytes for each type of addressing are shown in Fig. 5.13(b).

The SRA command operates in a less straightforward manner (Fig. 5.14) with bit 7 being copied into bit 6 (counting the lsb as bit 0) but no bit shifted in at the msb position — the original bit 7 remains in place. This command is seldom used.

The range of rotate commands of the Z-80 is rather daunting at first sight, and it's not much better at second sight. There are two directions of rotation, left and right, and two basic types of rotation, normal and

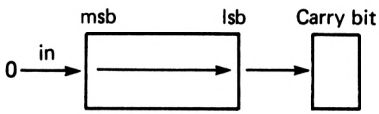
circular. The mnemonics RLCA, RRCA mean respectively rotate left circular accumulator, rotate right circular accumulator, and the mnemonics RLA and RRA mean the simpler rotate left accumulator and rotate right accumulator. Figure 5.15 summarises the effect of these

SLA r



Example: Register C contains 10010001, carry bit reset (0)
After SLA C the C register contains
00100010 and the carry bit is set (1)
The operand r can be any of the single-byte registers (Not R or I)
or (HL), (IX + disp) or (IY + disp)

SRL r

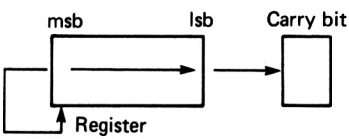


Example: Register D contains 10010001, carry bit reset (0)
After SRL D, the D register contains
01001000 and the carry bit is set (1)
The operand r is as above

Operand	*Bytes*	*Times*
A	2	8
B	2	8
C	2	8
D	2	8
E	2	8
H	2	8
L	2	8
(HL)	2	15
(IX + D)	4	23
(IY + D)	4	23

Fig. 5.13. The SLA and SRL shift commands. Above: both use the carry bit as part of the register whose contents are being shifted. Left: times and number of bytes for different addressing methods

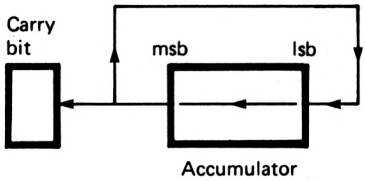
SRA r



Example: Register L contains 10010001, carry bit reset (0)
After SRA L, the L register contains
11001000 and the carry bit is set (1)
The operand r is as defined in Fig. 5.13

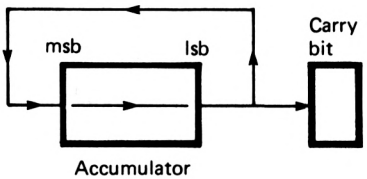
Fig. 5.14. The action of the SRA command

RLCA



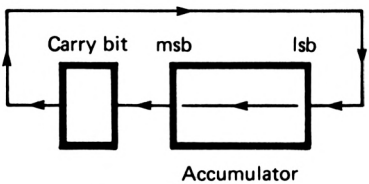
Example: 10011100 is stored in accumulator, carry reset (0)
After RLCA, 00111001 is stored in accumulator, carry set (1)

RRCA



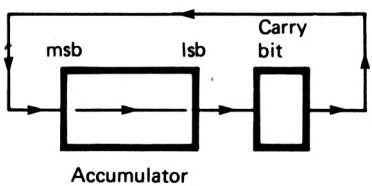
Example. 01001101 is stored in accumulator, carry reset (0)
After RRCA, 10100110 is stored in accumulator, carry set (1)

RLA



Example: 10011100 is stored in accumulator, carry reset (0)
After RLA, 00111000 is stored in accumulator, carry set (1)

RRA



Example: 10011101 is stored in accumulator, carry reset (0)
After RRA, 01001110 is stored in accumulator, carry set (1)

Fig. 5.15. The RLCA, RRCA, RLA, and RRA instructions and their effects

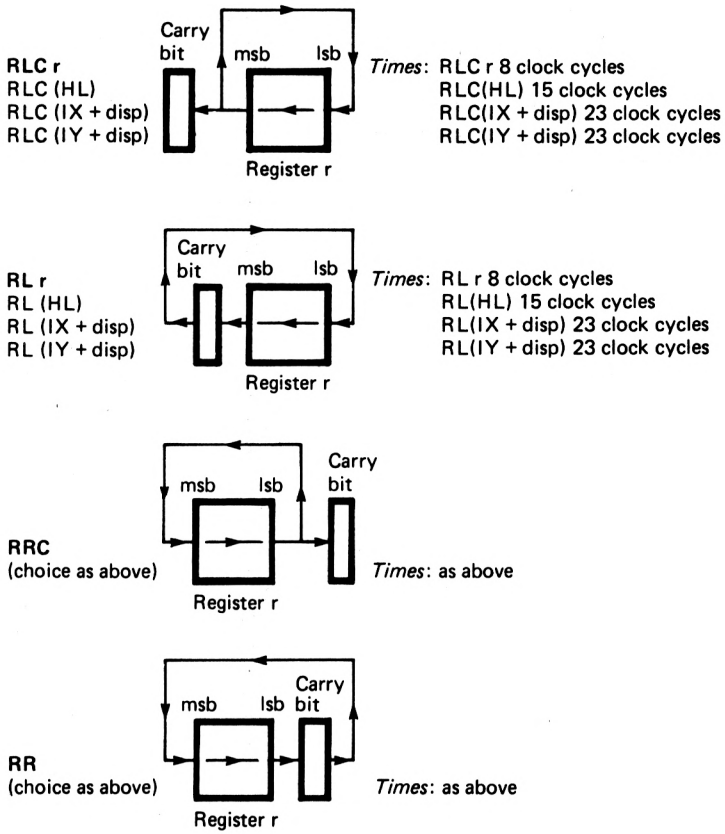


Fig. 5.16. The RLC, RRC, RL, and RR instructions used on other registers, with times

actions. Both the RLC and the RL operations can be carried out on bytes held in registers other than the accumulator (Fig. 5.16), and this is true also of the RR and RRC instructions. The number of bytes and the timing of these instructions are also shown in Fig. 5.16.

The Z-80 also supports a decimal rotate, RLD, RRD, operation whose action is rather specialised and of very little interest to most users.

Block instructions

A unique feature of the Z-80 as compared to many earlier designs is its

collection of 'block' instructions. A block instruction in this sense is one that carries out a set of actions, often automatically, which would otherwise need several lines of assembly language programming. A few of the block actions of the Z-80 will incorporate loops, so that they can repeat load and increment or decrement actions many times while a counter is decremented, ceasing the action when the count reaches zero. These features can make some types of programs remarkably simple to carry out on the Z-80.

```
        LD B, 26H
LOOP :  LD A, (memory)
        ADD C
        LD (HL),A
        INC HL
        DJNZ LOOP
```

The actions between LOOP and DJNZ will be carried out
26H times

Fig. 5.17. How a DJNZ action is programmed

The DJNZ command (Fig. 5.17) is often grouped with the jump set of commands, but its action is associated with the block commands. The B-register has to be loaded with a counter byte before the instruction is used, and on each occurrence of DJNZ, which has to be followed by a displacement byte, the content of the B-register is decremented and tested. If the byte in the B-register is not zero after decrementing, then a jump is taken to the address given by the displacement. This jump will normally be to an earlier part of the program, so that the DJNZ step can be repeated on each pass through the loop. When the count is finished and the decrement action results in the content of B being zero, the jump is not taken, and the program resumes with the next instruction following DJNZ. The instruction uses two bytes, one of which is a displacement, and needs 13 clock cycles when the jump is taken, eight clock cyles when the count is finished.

The other instructions in this group are instructions which transfer bytes from one memory location to another, and can be used in simple

```
LD HL, 36A2H      ; address in HL
LD DE, 4127H      ; address in DE
LD BC, Ø1Ø2H      ; count in BC
LDI               ; transfers byte in 36A2 to 4127,
                    increments both addresses, decrements
                    count
```

After LDI, the byte stored at 36A2H is copied to 4127H,
HL contains 36A3H, DE contains 4128H, and BC contains Ø1Ø1H

Fig. 5.18. The LDI instruction, showing the register loading and the action

form or in autorepeating forms, distinguished by the letter R for Repeat. The LDI (Load, Decrement, Increment) instruction (Fig. 5.18) requires the HL and DE registers to be loaded with addresses, and the BC register pair to be loaded with a count number. When the LDI instruction is encountered in a program, the byte held in the address in HL is transferred to the DE address, both of these registers are incremented, and the BC register pair is decremented. The only flag affected by this command is the P/V flag, which will be set until the number stored in the BC register pair equals 1; at this point the P/V flag is reset. The command needs two bytes and takes 16 clock cycles.

The automatic version LDIR carries out the same actions, but repeats automatically until the count in BC has reached zero. While the command is repeating, it needs 21 clock cycles, but on the last run it needs only 16 clock cycles. It is a two-byte instruction.

LDD	Copy byte from HL address to DE address. Decrement HL and DE. Decrement BC. P/V reset if BC = 1
LDDR	As above, but repeat action until BC is zero
CPI	Compare byte in HL address with byte in A. If equal, set Z-flag. Increment HL, decrement BC. P/V reset if BC = 1
CPIR	As above, but repeats until Z-flag is set or BC reaches zero. If BC = Ø at start of instruction, the instruction will continue to be executed for 65535 loops unless a match is found. S-flag set if result negative, Z set if a match is found, P/V reset if BC = 1, H set if there is a borrow
CPD	As CPI but HL and BC are both decremented
CPDR	Auto repeating version of CPD – as for CPIR but both registers decremented

Fig. 5.19. The LDD, LDDR, CPI, CPIR, CPD, and CPDR instructions summarised

This description of LDI and LDIR is enough to enable you to follow an abbreviated description of the other block-shift and compare commands LDD, LDDR, CPI, CPIR, CPD and CPDR, as shown in Fig. 5.19. Note that the use of R at the end of the mnemonic always means the autorepeat option.

Input-output commands

The input and output commands of the Z-80 are intended for use where the hardware of a system includes ports. The commands are of several types so as to allow the hardware designer the greatest possible flexibility in the use of these ports, and include several block commands which are seldom used by designers, mainly because their utility is limited to very fast transfers of data.

The simplest port input/output commands are IN A,(N) and

OUT(N),A respectively. The IN A,(N) command will place the operand byte which follows the instruction byte onto the lower half of the address bus of the system so as to select one of the 256 possible port addresses. The number in the accumulator is copied to the upper half of the address bus at this time in case the designer wants to make use of this; as, for example, in a matrix keyboard system. One byte from the selected port is then copied into the A-register, requiring a time of 11 clock cycles. The OUT(N),A command reverses this procedure, and requires the same form of command and the same number of clock cycles.

The IN r,(c) and OUT(C),r commands will load to or from any of the single-byte registers by using the register name in place of r, for example IN H,(C) or OUT(C),D. The port is selected by placing the byte stored in the C register on to the lower half of the address bus, and the byte stored in the B-register is put on to the upper half. One again, this byte in the B-register can be used as a gating signal for keys arranged in the conventional matrix. The instructions require one byte plus a port-number byte, a total of two bytes, and 12 clock cycles of time.

INI	Contents of C register placed on lower half of address bus to select port. Contents of B placed on upper half of address bus. Byte is read from port and placed in HL address. B register is decremented, HL register pair is incremented, Z-flag set when B = 1, reset otherwise
INIR	After INI action, if (B) is not zero, PC is decremented by two and the read action is repeated
IND	As for INI, but HL register pair decremented
INDR	As for INIR, but HL register pair decremented
OUTI	Byte is read from HL address. Counter byte in B register is decremented, contents of C register placed on lower half of address bus to select port. Contents of B register are placed on upper half of address bus, byte is then written to port, and HL register pair is incremented
OTIR	As OUTI; if B is not zero after action, PC is decremented by 2 and read-write action repeats
OUTD	As OUTI but HL is decremented
OTDR	As OTIR but HL is decremented

Fig. 5.20. The block port actions summarised

In addition to these commands, there are block input/output commands INI, INIR, IND, INDR, OUTI, OTIR, OUTD, OTDR whose functions are summarised in Fig. 5.20.

Miscellaneous commands

The huge instruction set of the Z-80 contains a lot of commands which, though useful at times, are not extensively used. Among these are the bit test and determine set, which allow any bit in a byte to be set (to 1) or

reset (to 0) or tested. The command, BIT 5,D, for example, will test bit 5 (counting from the lsb at bit 0) of the byte in the D-register, and alter the Z-flag to signal the result. Any of the bits numbered 0 to 7 can be tested, and any of the single registers can be used to hold the byte. The commands use two bytes and take a time of eight cycles. The test can also be carried out on a byte whose address is held in HL (two-byte command, 12 clock cycles) or on a byte whose address is held in the index registers, in which case the command consists of a total of four bytes and requires 20 clock cycles.

The SET command also operates on bits selected from a byte, so that SET 2,L will, for example, ensure the bit 2 of the byte in register L is 1. The same range of bits and registers is available, as well as the choice of HL or indexed addressing. The corresponding RESET command, RES, follows the same pattern of addressing and timing.

Some of the other commands are less frequently used. DAA is a decimal adjust command, which will correct the value in the accumulator after an addition of BCD digits. Normal binary arithmetic carried out on bytes that represent BCD numbers will give an incorrect result unless this command is used. NOP is the no-operation command, during the course of which the Z-80 will continue to refresh any dynamic memory but will not perform any instructions. This is used either as a time-waster (four clock cycles) or as a way of leaving space in a program for inserting instructions later. Its opcode is ØØ, so that if memory has been zeroed the presence of a piece of unused memory in a program will not cause the execution of unwanted commands.

HALT is used to suspend operation indefinitely, though the refreshing of dynamic memory continues. The Z-80 remains in its halted state until an interrupt or a reset instruction is received. DI means disable interrupt, and will ensure that the Z-80 cannot be interrupted on the maskable interrupt pin after this instruction has been executed. The

CPL	Complement accumulator. Exchange each 0 for a 1 and each 1 for a 0 in accumulator Time 4 cycles, single-byte instruction H and N set, other flags not affected
NEG	Convert contents of accumulator to negative (2's complement) form. The byte 8Ø will be unchanged by this command Time 8 cycles, two-byte command N set, all other flags affected
CCF	Complement carry flag. Invert carry bit Time 4 cycles, single-byte instruction S,Z,P/V not affected. H copies previous carry, C inverted
SCF	Set carry flag Time 4 cycles, single-byte instruction S,Z,P/V not affected. H reset. C set

Fig. 5.21. Some of the miscellaneous commands summarised

non-maskable interrupt is still operative, so that a closed loop executed following a DI instruction can still be broken if the hardware permits a reset switch on the NMI line. DI is used when the Z-80 is about to execute a routine that involves precise timing, such as disc or cassette saves or loads, so that these routines cannot be interrupted by pressing any of the computer keys. At the end of a routine which has started with DI, EI can be used to re-enable interrupts.

Other instructions which are seldom used are listed, with brief descriptions, in Fig. 5.21.

6

Program design

Every writer on assembly language offers advice on the design of assembly language programs; some even offer a choice of different types of advice. The reason is that no single set of mechanical steps can be guaranteed to lead to programs that will fit your specification, because so much depends on what you are looking for. Many prospective writers of assembly language are looking for a solution to a problem that cannot be tackled adequately in BASIC, and are perfectly satisfied with a machine-code routine that works. Designers of machine-control systems are concerned with the interaction between the microprocessor system and mechanical systems, and their first priority is a foolproof program, one that never permits the machine to run out of control. Designers of compilers and interpreters (even the most modest of such programs) are concerned with writing the most compact code that will achieve their objectives without taking up too much memory space. Authors of games written in machine code are generally more concerned with fast-running code rather than with compact code, so that the animation of graphics is reasonably lifelike (or monster-like, as the case may be).

There is no perfect overall way of designing assembly language programs that will satisfy all of these demands in exactly the same way, hence the variety of advice that is offered. One method, however, will serve to get you started on whatever type of program you write, though how you tackle the details will depend on your precise requirements and on the extent of your experience. In addition, since this method is essentially the same as is used in the design of BASIC programs, you may feel your feet to be on familiar ground.

The place to start, then, is at the top. The essentials at this stage are what keys, if any, you need to press to start things off, whether there is any disc or cassette input or output, and what result you expect, either in memory, on the screen or on paper, disc or cassette. If you are designing for a Z-80 based computer, the result may be something that appears on the screen; if you are designing a program to operate a robotic arm, then the result will be the movement of the arm. Don't neglect this 'master-plan' step, because this forms the description of the program that you will have to refer to as you develop the program. If, as design progresses,

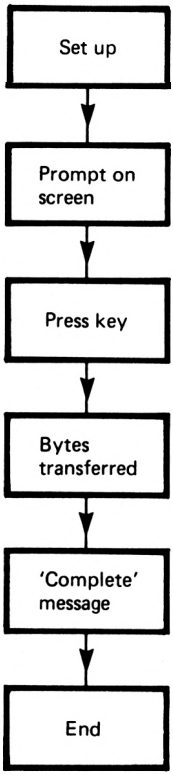


Fig. 6.1. A simple 'master plan' for a program routine. This is not a true flowchart, simply an indication of sequence of action

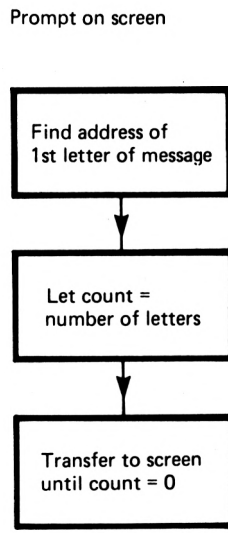


Fig. 6.2. Breaking down the plan. Each part of the plan, as illustrated here for the 'prompt on screen' section, can be broken down in more detail

you think of improvements that can be made to this scheme (and you will!), then make sure that a new version is drawn up. Figure 6.1 shows an example of a simple master-plan of this type. If this plan is inadequate, then you will be greatly hindered as you try to move to the ultimate stage of writing assembly language. Rule 1 is that you have to know what you want to write! Don't assume that you know in your mind's eye what you want and that all you need to do is to turn out code. After a few sessions of assembly language you can get so immersed in the details of assembly language that you completely forget what your original intention was.

The next step is to start breaking up the overall plan into steps which can be executed one after the other. This does *not* mean assembly

language unless the master plan was for a very elementary program, only that the scheme should be broken down into smaller steps, as illustrated in Fig. 6.2. At this stage some designers like to use flowcharts, others prefer less formal ways of indicating the order of steps and the type of step that is used in each case.

The aim now is to keep repeating this breakdown process on each step until we reach a stage where we have steps that can be carried out by assembly language subroutines, as shown in Fig.6.3. These subroutines

```
LD HL, WORDS
LD BC, COUNT
LD DE, SCREEN
LDIR
```

Fig. 6.3. The final stage — an assembly language subroutine

may be short, like an LDIR block shift, or considerably larger, such as we might find if a subroutine from the ROM of the computer were used. At some stage in this breakdown process, we must be able to recognise where a machine-code subroutine can be used, and this is the stage at which experience, rather than book reading, is the only useful guide. The more you work with assembly language, the easier this stage becomes, because you start to recognise familiar situations for which you know a subroutine. At this point each section of the program should be a complete entity, with one starting point and one finishing point, and your plan should show what conditions exist at the start and end of each section. Suppose, to take a very simple example, that your program has aimed at transferring a diagram on the screen into printed form. We might have evolved the overall plan that is shown in Fig. 6.4. In this plan we assume that the block graphics that appear on the screen are memory mapped in a straightforward way. This means that each position on the screen corresponds to a different memory address, and that going through the memory addresses in sequence will correspond to moving across and down the screen in an equally orderly fashion. This assumption is true of a number of computers, notably the TRS-80/Video Genie series, but is certainly not true of some recent designs, particularly the ZX Spectrum.

The second assumption concerns what is printed. Some printers, such

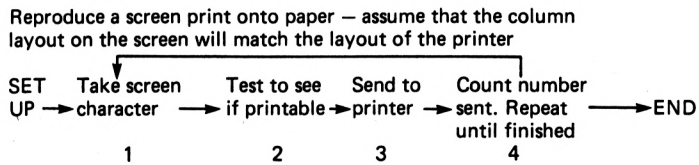


Fig. 6.4. A sample plan, showing a different type of format. This is less formal (and less tidy!) than a flowchart

as the EPSON MX-80 can print 'dot graphics', meaning that they will print shapes that can be programmed byte by byte. Others will not, but can (as can the EPSON also) produce a range of standardised blocks when the appropriate code numbers (usually above 8ØH) are sent. Very often, the printer that is recommended for a computer can print all the low-resolution graphics characters of that computer — a case in point is that the EPSON can be switch-set to print the TRS-80 graphics characters. We shall assume that the printer will print the graphics character shown on the screen when the same code is sent to the printer. The second step of our plan, however, checks the size of this code number to make sure that it is not one which can cause unwanted effects, such as changing print size.

This example is a simple one, so that breaking this down into assembly language subroutines is also easy, but a few points of detail must be added. How, for example, is the computer to be instructed to carry out this action? What message should appear when the action is finished? These problems presuppose a detailed knowledge of how machine-code programs are run and how messages can be printed, and the solutions require you to know the workings of your own machine well — there is no general solution that I can give which will suit all needs.

We shall concentrate on the main parts of the scheme, therefore. The program is a looping one, which involves a countdown, and it looks as if we could use several of the double registers of the Z-80. Before we can start writing assembly language, however, we need to examine each section in detail, and this is where the remarks made earlier on starting and finishing conditions can be explained.

Taking the first block, we need a starting address at which the first graphics block on the screen might be found. This point corresponds to the top left-hand corner of the screen on many computers, though some designs take it as being the bottom left-hand corner. We can call this address SCRNST, meaning screen-start. The restriction to six letters is deliberate because many assemblers only permit a maximum of six letters for labels. When we come to write the program, we can fill in this address, using a line such as

```
SCRNST EQU 3CØØH   (for example)
```

We also need a count to ensure that we send to the printer only as many bytes as would make a complete screen picture. This count is likely to be the maximum possible number of screen bytes, less 1 (since we start at the first byte), and we can label this number as COUNT. Once again, the value of this label can be assigned by an EQU statement at the start of the assembly language program. If we need to change it, so that the program will be usable on another Z-80 based computer, it's a simple matter to change the EQU statements, and we know where they all are.

The first stage looks as if it is simply a matter of loading a byte from an address. The general custom in Z-80 programming is to use HL to hold addresses, BC to hold counts of more than one byte, and the accumulator to handle single bytes. We should therefore set up the conditions before the loops starts, by statements such as:

```
LD HL, SCRNST
LD BC, COUNT
```

which will get these quantities into the correct registers. At this stage we'll simply note that these are the registers used for storing these quantities, and the detailed description for this section is therefore as shown in Fig. 6.5. The main action will be just

```
LD A,(HL)
```

which will get the byte at the screen address held in the HL register pair into the accumulator where we can use it.

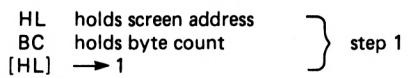


Fig. 6.5. A statement of the set-up conditions for the program

The illustration shows the conditions; HL contains the current screen address and BC contains the count. This is true at all times when this stage is executed, but the start of the program will have inserted the quantities, and we have already dealt with this stage, the set-up stage.

The next section deals with which characters can or cannot be sent to the printer. The character picked from the screen memory is now in the accumulator, so that we can test it by using the CP instruction. What we need now is a specification of what characters must *not* be sent to the printer, and what we do if the accumulator contains such characters. This is one of the items that should have been considered during the master-plan of one of the early break-down stages. Figure 6.6 shows a possible treatment, assuming that no character codes less than 20H (the ASCII space character) can be transmitted, and that if such a character is encountered a space will be sent. This is, once again, a condition that I

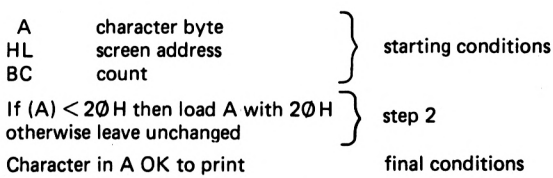


Fig. 6.6. The second section of the program broken down

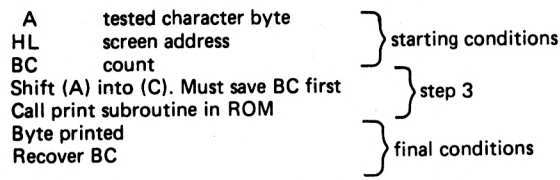


Fig. 6.7. The third stage — if the PRINT subroutine requires the byte in register C, then the contents of BC will have to be saved on the stack

have assumed — a few computers do store such characters in the screen memory. At the start of this stage, therefore, the character is in A; at the end of the stage a character, which may not be the same character, is in A.

The next stage involves sending the byte to the printer. When you are using a Z-80 based computer, there will invariably be a routine in the ROM which will cope with sending a byte to the printer. If both printer and screen use ASCII codes, this routine will be straightforward; less so if the coding of the screen memory is not ASCII. In some cases, the printer may need a special subroutine which has to be written and placed in RAM — but that's another problem! The point here is that you can't assume that the printer routine that you use will require the byte in register A. Just to emphasise the point, Fig. 6.7 assumes that the character has, in fact, to be in the C-register, so that a command LD C,A will have to be used in the routine at some point. The business of printing the character can then be left to the ROM subroutine by having a CALL to its address number.

There is a snag here, however. If we transfer the byte in A into the C-register, we shall corrupt the counting process, because we are using BC to hold the count. The BC register contents will have to be saved from this possible corruption by placing them into memory, and we have to make a note to do this before we write the assembly language.

The next step is the crucial one of deciding whether or not to go back and do it all again, with an incremented address. The steps at this stage are shown in Fig. 6.8; the screen address is incremented, the count number is decremented, and the count is tested to see if we have come to

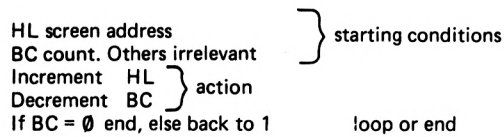


Fig. 6.8. The action after printing — the program must now check the count to decide whether another byte has to be transferred

the end of the screen memory. If this is so, the routine must halt, returning to the computer operating system; if not, then the routine must loop back for another character byte. The complication here is that the BC registers must be recovered from memory before the count can be decremented or tested, because at the time when we start this part of the routine, C contains the byte that was sent to the printer, not the count byte.

This breakdown leaves us almost ready to start writing the assembly language version. One minor point at this stage concerns interrupts — do we want the screen dump to proceed to the bitter end regardless of any attempt to stop it? If so, then we shall have to start the program, just before the loop section, with a DI instruction, and place an EI after the program finishes the loop. If, as is more likely, we want to be able to change our minds about tying the computer up for a time in this printing operation, then we can omit the DI, EI pair of instructions.

We are now ready to write the full version of the assembly language. This is shown in Fig. 6.9 in full Z-80 assembly language format, but with the stage numbers corresponding to those shown in Fig. 6.4 added at the side. A run through this simple routine may be useful as a way of pointing out the methods of assembly language programming that are not fully brought out by a recital of the available commands.

```
1.          ORG 7FFØH
2.          SCRNST : EQU
3.          COUNT : EQU
4.          LD HL, SCRNST
5.          LD BC, COUNT
6. LOOP:    LD A,(HL)          ;get character
7.          CP 2ØH             ;compare to 2ØH
8.          JR NC, OK          ;NC if A more than 2ØH
9.          LD A, 2ØH          ;put blank in otherwise
10. OK:     PUSH BC            ;save count bytes
11.         LD C, A            ;get in correct register
12.         CALL PRINT         ;print it
13.         POP BC             ;get count back
14.         INC HL             ;bump up address
15.         DEC BC             ;countdown
16.         LD A, B            ;test....
17.         OR C               ;for end of count
18.         JR NZ, LOOP        ;if not ended, back
19.         RET                ;ended, so return
```

Note the label PRINT, which means the address of a ROM routine would have to be assigned, or the actual address specified at this point. Since the routine address is not likely to be changed, it is better to fill in the actual address at this stage.

Fig. 6.9. The assembly language program. This assumes that the format of the printer (columns per line) matches that of the screen. Appendix A shows what is necessary if the two are not identical

The set-up stage is one we have already dealt with. The address of the start of the machine codes will have to be placed in an ORG statement, and the value of the screen memory address and the count size placed in EQU statements. These three statements are pseudo-instructions; they are not assembled into code like real instructions, they are simply used as instructions to the assembler program itself. The code starts with the loading of HL and BC in the fourth and fifth lines of the program. These actions are carried out once only, and the loop starts in the sixth line. The label word 'loop' is used preceding the command to identify the memory address so that the jump-relative command near the end of the program can return to this point for another load from screen memory. This single command carries out the requirements of Step 1, getting the character from the screen address into the accumulator for testing.

Lines 7 and 8 carry out the test-and-jump steps. By using CP 2ØH, we set the flags that would be set by SUB 2ØH, but without affecting the byte in the accumulator. Now if the byte in the accumulator is less in value than 2ØH, such a subtraction would require a 'borrow', and this would set the carry flag. If the byte is equal to 2ØH, or is greater than 2ØH, the carry flag will not be set. This is a more useful test than using the zero flag, which can check only if the byte is equal to 2ØH. The Z-flag and the C-flag are the only ones that are tested by a JR jump, so that we can't take the more obvious step of testing the sign flag unless we use the JP type of jump. There is no reason in a program of this type to exclude the JP jump, except that it needs one more byte and will make the program non-relocatable — that's a matter of choice.

If there is no carry, meaning that the byte is 2ØH or more, then the jump in line 8 leads directly to line 10. If there has been a carry, then the program proceeds to line 9, loading the accumulator with 2ØH in place of the byte that was there originally. This done, the next line is 10, which pushes the BC register pair on to the stack. We need to put the byte in the accumulator into the C-register, but we don't want to change the count that is held at present in BC, so the present bytes in BC have to be preserved in the stack memory until we need them again. We can then prepare for printing the byte by transferring it from the A-register to the C-register since that is what our (imaginary) print routine requires. The label OK is used to allow the assembler to insert the correct displacement for the jump in line 8. It's tempting to ask why the byte should be transferred — why not use LD C,(HL) in line 6? The answer is that the CP step needs one byte in the accumulator, so that it nearly always pays off to put into the accumulator any byte that will be manipulated in any way.

Having loaded register C with the byte, the print routine in the ROM (or wherever it is located) is called into action. This should send the byte to the printer, with the correct timing, and then return to perform line 13

of our program, which restores to BC the original contents of registers B and C from the stack. The program then carries out line 14, which increments the HL address. I have assumed that the screen memory uses ascending address numbers. If the numbers are in descending order, which is unusual, then DEC HL would have to be used at this point. In line 15, BC is decremented, and lines 16,17 and 18 test for the end of the count.

As was explained earlier, decrementing BC to zero does not affect any of the flags, so that if a decrement of this type has to be tested, it must be done in a roundabout way, using the OR C step as the test. The POP BC action will replace the byte in register C by the count lower-byte, and the LD A,B step will similarly replace the byte that was stored in A, but since the byte has been printed by this time, its replacement is not a problem. The flag setting is then tested in line 18, and if the zero flag is not set (count not zero), then the program returns to the LOOP address. If the action is complete, then the program returns to the machine operating system.

Though this is a simple program it illustrates a large number of points about assembly language programming with the Z-80. One point is that the amount of paper used in pre-planning is greater than the amount used for writing the assembly language! Another valuable point is the extent to which the accumulator has to be used because so many commands affect only the accumulator. A third point is the use of PUSH and POP commands for the stack, commands which are dealt with in more detail in the next chapter. Though the Z-80 can use a large number of registers, this is never as many as the number of possible BASIC variable names that can be used in a BASIC program. Very great care must therefore be taken whenever a register is used that either the byte stored there is no longer needed, or that the register contents are saved on the stack or otherwise into memory. Overlooking this point is one of the commonest causes of program failure, and it isn't something that just happens to beginners!

The program has also demonstrated the use of label names, both in loops and in assignments of address and count numbers. The principles of design and construction that we have used here should form a good basis for your own programs, providing that your ambitions do not exceed your experience by too great an amount!

Inputs and outputs

The example program bears out the problems of dealing with inputs and outputs. When a program is being written for use in an existing computer, the input/output routines of that computer should be used as

far as possible, because writing your own routines is tedious, generally unnecessary, and demands a good understanding of the hardware as well as of programming. The only case when you might have to write your own routine is when suitable computer routines do not exist. A common situation is when a computer does not have a serial RS-232 input or output, and one is added in hardware. A program has then to be written which will convert data into the form needed for RS-232 transmission into the hardware circuits, and such a program will inevitably make use of the port instructions. For most computers which do not have port circuits built in, add-on ports, some of which will implement all 256 possible port addresses, are available.

When code is being written for a new piece of hardware, such as a robotics system, which is not a part of an existing computer, then the whole of the software, including the input/output routines, will have to be written from scratch. This includes the correct setting up of the port chip as well as the transfer of bytes between the port and the accumulator. This can be very tedious, particularly if the routines are being written for an unfamiliar type of port chip. Close attention to the specification of the chip, as detailed in the manufacturer's literature, is essential. One point that I have found useful is to copy out the relevant parts of the port specification. Since most port chips are multi-purpose chips, it can be very confusing to wade through the manual that comes with the chip, trying to sort out the relevant parts. While this is being done, the relevant parts, and only the relevant parts, should be copied. This way you end up with a guide to setting up the port in the way you want it, without having to separate this information from the parts you don't want or need.

7

Programming details

This chapter deals with the details that have been omitted in previous chapters because we have been dealing as far as possible with the instructions that are most widely used. Chapter 5 dealt in detail with the Z-80 instructions, arranged in groups, so this chapter is concerned with methods of using instructions rather than with the details of the instructions themselves. We shall start with the use of the stack.

The stack

The stack is the name given to a part of the RAM which can be selected by the programmer. The starting address for the stack should be at the top address of a section of RAM which will not be used in any other way. A convenient place is very often the highest address in available RAM, so that an instruction of the type:

```
LD SP, RAMTOP
```

will place the stack at the address which lies below the selected starting point, labelled as RAMTOP, a number that will have to be allocated by EQU or in the other usual way.

The stack pointer is a 16-bit register which is used purely to hold stack addresses, and the convention is that the stack pointer always holds the address of a byte that will be returned to a register when the stack is in use; when the stack is not in use it will hold the top-of-stack address. For most purposes when you are using an existing computer, you will be able to use the existing stack; it is seldom necessary to change this set of addresses. One exception is if the machine code that is to be assembled will take up the memory space that is normally occupied by the stack, in which case the machine code will have to allocate a new address for the stack pointer before any attempt is made to place machine code into memory.

Once a stack starting address has been allocated, any operation that requires the use of the stack will automatically make use of an address below this starting address, changing the stack pointer in the course of

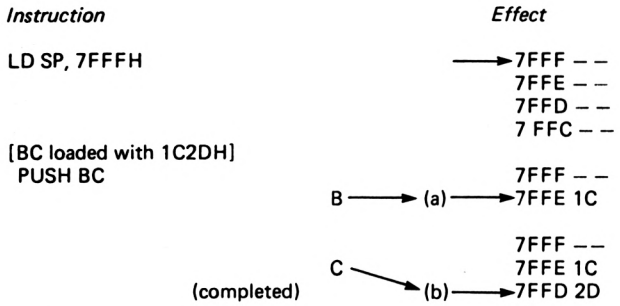


Fig. 7.1. The PUSH instruction, which places two bytes on to the stack memory. Note that the stack pointer register is decremented *before* each load

the action. A PUSH operation, for example, starts by decrementing the stack pointer (SP) register, and then loads into the memory at the SP address the high-order byte from the register pair — the contents of the first register of the pair (such as B,D,H). The SP is then decremented again, and the low-order byte is loaded in, leaving the SP storing the address in RAM of this low-order byte. This process is illustrated in Fig. 7.1 — note that the address of the start of the stack is not used at any time; when the SP contains this address, the stack is empty. One of the main problems with the use of the stack is the number of misleading terms that are used, such as 'top of the stack' to mean the current SP address. Throughout this book, I shall use the terms 'current SP address', 'push on to the stack' and 'pop from the stack' to indicate the stack actions.

A POP operation is illustrated in Fig. 7.2. The low-order byte is read from the current SP address. The SP is then incremented, and the high-

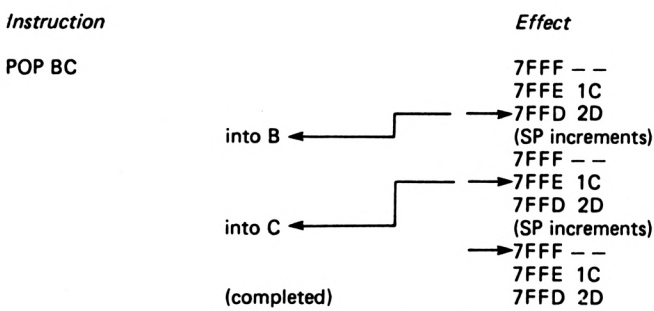


Fig. 7.2. The POP instruction, which copies bytes from the stack into nominated registers. Note that the stack pointer is incremented *after* each load

order byte is read. The SP is then incremented again, so that its current address is either that of the next low-order byte or the address that was originally allocated to the SP as its starting address.

The stack is used as a form of serial memory, in which the last byte written in is the first byte to be read out. The advantage of stack use is that it needs generally only single-byte instructions, and 11 clock cycles to push a register pair, as compared with actions such as LD(NN),HL which need three bytes and 16 clock cycles. The fact that the stack pointer stores the register contents strictly in order is seldom a disadvantage, and, if necessary, the stack can be used as a form of random access memory by changing the SP address (by the use of LD SP,NN or by INC SP) and then accessing a stored byte by using EX(SP),HL to get the contents of two SP addresses (formerly register contents) into the HL registers.

In normal use, then, registers are saved on the stack (written to the stack) by commands of the form:

PUSH AF (or BC,DE,HL,IX,IY)

which saves two bytes on the stack. Note that when AF is saved, the conditions of all the flags in the flag registers are being saved, and these flags will be restored when a POP AF command is executed. Some care has to be taken with conditional jump commands that are used near a POP command, because you have to be certain that the flags which affect the jump are the ones you expect to affect it. For example, if you have a JR NZ jump immediately before a POP AF, then the flags which affect the jump are these set by the most recent operation that affected flags, which might be an OR A or something similar. On the other hand, if the JR NZ occurs immediately after a POP AF, then the flags which affect it will be the flags which were set just before the AF registers were pushed; an operation which might have occurred some considerable number of steps back in the program.

In normal operation, each PUSH action will be reversed by a POP action later in the program. The POP action will place two bytes from the stack, at the current SP address and the next higher address, into a chosen register pair. Note carefully that the register pair into which a pair of bytes is placed by the POP action need not be the register pair that was used by the PUSH action. There is nothing automatic about this; POP simply means that two bytes are read from the stack into a register pair and there is absolutely no guarantee that they are the correct two bytes — that's up to you, the programmer. Figure 7.3 shows the routines that are needed to save all the normal register set on to the stack and subsequently recover them again. Since the stack works on the first-in, last-out principle, the order of POP operations must be the opposite of the order of the PUSH operations if the registers are to be correctly

```
PUSH AF
PUSH BC
PUSH DE
PUSH HL
PUSH IX
PUSH IY
[subroutine]
POP IY
POP IX
POP HL
POP DE
POP BC
POP AF
```

Fig. 7.3. Saving all the normal registers (not alternates) on the stack Note that the order in which the bytes are POPped must be the opposite of their PUSH order

restored. This order can be changed if there is a good reason for doing so. The actions:

```
PUSH DE
POP BC
```

very neatly place the contents of the DE register pair into BC, so that to exchange the contents of DE and BC, we need

```
PUSH DE
PUSH BC
POP DE
POP BC
```

to effect the exchange. There is a command EX HL,DE which will exchange these two registers, but there is no such automatic exchange command for HL and BC, or for DE and BC, to take just two examples.

The stack is used automatically by the CALL and the RST instructions, which place the PC address on to the stack before executing subroutines. As we noted earlier, when we save other registers on the stack in the course of a subroutine, these register contents must be popped off the stack before the RET command is executed, else the return will be to the wrong address. This is sometimes done deliberately as a way of causing a return to some other address, but it is not in any sense programming for novices!

Any beginner to the use of Z-80 assembly language is advised to keep to the straight and narrow path of conventional PUSH and POP until the use of the stack has become familiar. A read through the dissassembled listings of the ROMs of Z-80 based computers often reveals interesting uses of the stack which can cut several bytes from a routine; but these tricks should be left severely alone in the beginning. The instruction set of the Z-80 permits a lot of manipulation of the stack pointer, but it doesn't follow that you *have to* use them all.

The alternate registers

The alternate registers of the Z-80 form an alternative to the use of the stack for the temporary storage of bytes, because the exchange commands for the alternative registers consist of only one byte and take only four cycles. A problem that calls for the use of the alternate registers, for example, is typically one where all of the main registers are in use, but a short manipulation of one byte is needed in the accumulator. This can be programmed by using:

```
EX AF,AF'              ;put alternates in place
LD A,20H               ;routine starts
. . .carry out routine . . .
EX AF,AF'              ;back as before
```

When this is done, the bytes that were stored in AF are unaffected by the interchanges and will be present again in these registers after the second exchange. Between the two occurrences of EX AF,AF', any instruction that makes use of A or F will use the alternate registers rather than the main registers.

EX AF,AF' (ZEN uses the operand AF,AF for this example), will exchange the AF registers with their primed equivalents in the alternate set, and these only. Another command, EXX, has the effect of exchanging all the other alternates, BC',DE' and HL'. This releases the use of BC, DE and HL registers for other purposes such as block instructions, and the normal registers, with their contents, will be restored when the second EXX command is executed.

Indexing

Anyone who has used indexed addressing on the 6502 will find the Z-80 indexing system curiously restricting in comparison, despite the use of 16-bit registers. If the instructions were of the form LD A,(IX + C) for example, where C represented the byte in the C-register, the system would be very much more powerful and flexible. As it is, the use of indexed addressing is rather limited, but that does not mean that programmers do not make good use of it, as a glance through a disassembly of the ROM of the ZX Spectrum will reveal.

In general, indexed addressing is used when two corresponding values have to be accessed, or where a set of values is held in memory to be used at intervals in a program. Taking the second application first, imagine that a program for a robotics device uses a set of 12H parameter bytes for defining operation conditions. These bytes can be held in sequence in memory starting at some RAM address which for the sake of example we

can take as 7FØØH. Any of the bytes can then be accessed either for reading or writing by specifying its displacement number from this starting address, providing that one of the index registers, IX for example, has been loaded with that address early in the program. In such an example IX would be loaded with 7FØØH early in the program, and this address would remain in IX throughout the whole of the program. This contrasts with our normal use of HL and other register pairs, in which we expect to change the contents of registers at intervals throughout the program. To access a byte, using indexed loading, we would then make use of commands such as LD A,(IX + 5), which in our example would load in the byte in address 7FØ5H or LD (IX + ØCH), A which would load the byte in the accumulator into address 7FØCH. There is, on the face of it, no memory or time advantage in this method of loading, but the convenience of being able to keep a fixed address in a register and refer to bytes by using displacements is often very useful.

Taking the other use of indexing, if we have a set of bytes located starting at some address such as 7FØØH and another set starting at some other address such as 6FØØH, then a byte loaded from an address using LD A,(IX + 5) can be matched with one loaded using LD B,(IY + 5), where IX and IY have been loaded respectively with 7FØØH and 6FØØH.

Port input and output

The port of a microprocessor system is the circuit or circuits through which the microprocessor makes contact with circuits beyond the microprocessor system. A port should not be confused with interfaces, though in many cases the port is used to perform some interface actions. In general, a port will deal with the data bytes that the microprocessor will handle, along with synchronising signals of the form generally referred to as 'handshakes'. The port can store bytes in its own registers, so that an output from a microprocessor whose duration may be only one clock cycle can be held at the port in a data output register until it can be used. Similarly, an input to a port can be held in a data input register until the microprocessor is free to read it. These signals are, however, normal + 5 V and 0 V signals, and if we want them converted to other voltages, or into analogue signals, or into signals that can provide more than a few microamps of current, then separate interface circuits will be needed. The port deals with the timing and selection of signals rather than with their size.

A port of sorts can be constructed from latch chips, and for many purposes a simple latching port (latching means that signals can be stored) is adequate. Semiconductor manufacturers, however, make a

large range of port ICs, and because the cost of making an IC bears very little relationship to the complexity of the circuits in it, the ports which are available in IC form are very complicated devices, rivalling the microprocessor itself in terms of physical size, number of pins, and number of registers. The idea has been to make the port chips as flexible in use as possible so that designers could use the same chip irrespective of how simple or complex their requirements are. Unfortunately this has led to the design of such complicated ports that many hardware designers are discouraged from using them.

It is no part of the brief of this book to examine how ports are operated and connected into circuits, but we need to spend some time discussing the program methods that we use. Chapter 5 noted the use of the commands IN and OUT which relate to port operation, and in this section we shall look at the problem of configuration of ports.

Configuration means setting up the control systems of the port so that its inputs and outputs are in the required form and at the correct pins. The Z-80 parallel port (Z-80 PIO) (Fig. 7.4) uses two 'ports', groups of eight input/output lines and two synchronising (control) lines which can be connected to circuits outside the microprocessor system. The use of the word 'port' to mean a group of input/output pins as well as to mean the complete interface can be confusing, and I shall use the words input/output or the abbreviation I/O to mean the actual pins which supply or accept signals. Each of these two I/O sections (Fig. 7.5) is configured by a set of five registers, and will also use two data registers, one for inputs and the other for output. The configuring registers (not all eight bit!) are named and carry out actions as follows:

1. *Mode control.* The number stored in this register decides whether the I/O is to be used with all pins as inputs, all pins as outputs, all pins bidirectional (both input and output), or each pin separately controlled as an input or output. There are four choices here, so that the mode-control register needs two bits only.
2. *The input/output control register.* The setting of the bits in this register will decide whether the pins of a port are used for inputs or for outputs when the mode-control register is configured for separate control. This is an 8-bit register, since the status of each bit decides whether the corresponding pin is an input or an output.
3. *The mask-control register.* This is another 2-bit register which determines whether a 0 or a 1 will count as an active signal, and whether an interrupt will be generated by ANDing or ORing signals. This register is used only when pins are individually programmed as inputs or outputs.
4. *The mask register.* This consists of eight bits and is used to decide which bits in the I/O will be used to carry an interrupt signal. Once

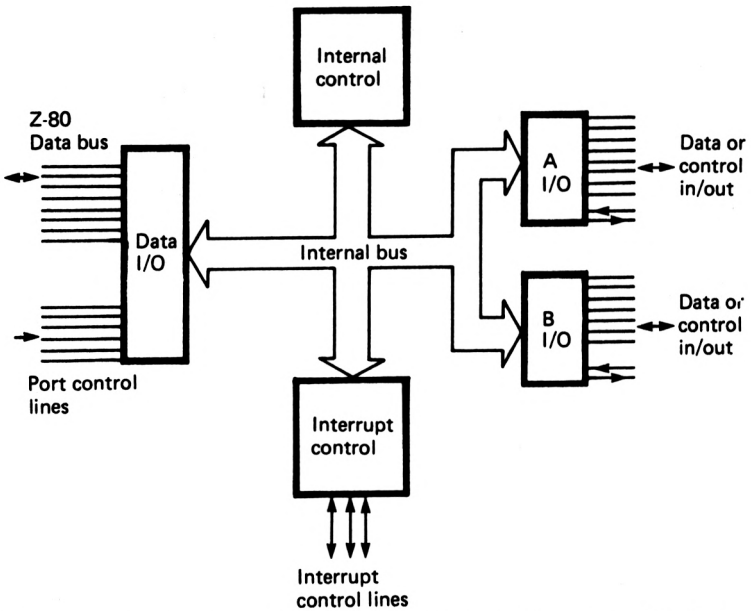


Fig. 7.4. A block diagram of the Z-80 PIO (Parallel-Input/Output) port chip. This provides two sets of 'ports' or input/output lines

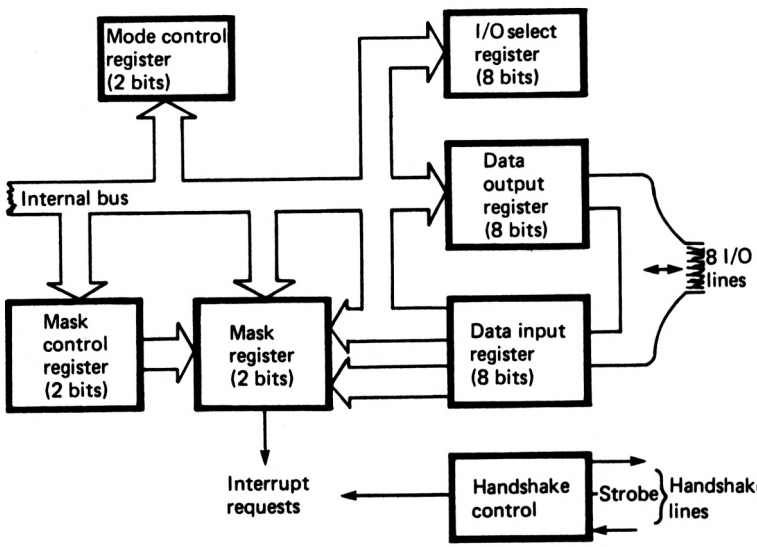
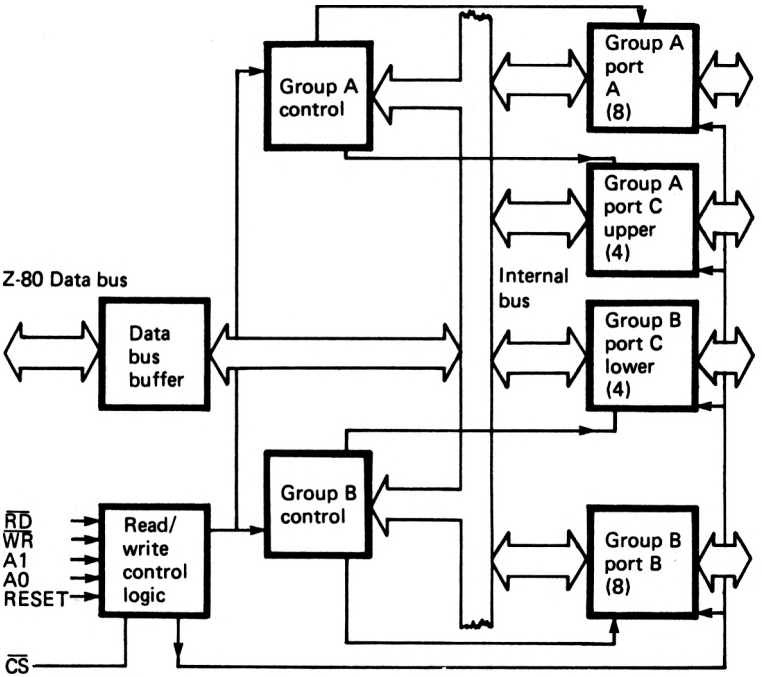


Fig. 7.5. A block diagram for each port section of the PIO, showing registers which have to be written to to configure the port



Selecting signals

A0	A1	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	Operation
0	0	0	1	0	Read port A in
1	0	0	1	0	Read port B in
0	1	0	1	0	Read port C in
0	0	1	0	0	Write port A out
1	0	1	0	0	Write port B out
0	1	1	0	0	Write port C out
1	1	1	0	0	Write to control registers
X	X	X	X	1	Disabled
1	1	0	1	0	Illegal – this set of signals must be gated out
X	X	1	1	0	Disabled

(X – either 0 or 1)

Fig. 7.6. The 8255 port block diagram, with details of some of the selecting signals. Lines A0 and A1 connect to the lowest order address lines of the Z-80, and the chip is activated by a signal on the CS pin

again, this is used only when the pins have been individually configured.
5. *The vector address register.* This is used to supply an address to which the microprocessor will jump when an interrupt occurs.

In general, a port is configured only once in the course of a program, and the remaining use of the port concerns only port inputs and outputs, with any interrupt signals that pass through the port. The configuration of a port would be perfectly straightforward if each register were allocated a separate address, but this is seldom the case, so that very close reading of the manufacturer's specification is needed before the assembly language configuration program is used. Just to quote one example, all of the control registers of the Z-80 PIO are accessed through one address, so that the data bits that are initially sent to that address will also be used to select the other registers in sequence.

Figure 7.6 shows an outline description of the INTEL 8255 port chip as used in MENTA.

Interrupt systems

Whether a port is used or not, some understanding of the Z-80 interrupt system is needed, either from the point of view of designing machine-control systems or from the very different viewpoint of understanding the operating system of a computer. An interrupt, just to recall and summarise, is a 'hardware' signal, a logic change of voltage at one of the two interrupt pins of the Z-80, which causes the program to be interrupted. This means that the normal program action will be suspended while the microprocessor attends to other actions. These other actions will be controlled by a machine-code program whose starting byte is located at an address which is called the *interrupt vector.* The machine-code interrupt routine is often called the *'interrupt service routine'.* One essential part of the interrupt service routine must be to save the contents of registers on to the stack so that the Z-80 can resume normal action again after the interrupt has been dealt with.

The two interrupt pins of the Z-80 are known as the maskable interrupt (MI) and the non-maskable interrupt (NMI) respectively. Of these two the maskable interrupt has the lower priority, and any interrupt signals to this pin will be ignored if the software instruction DI has been executed. The use of DI will disable the MI pin until the EI instruction is executed or until the microprocessor is reset by a signal on the RESET pin.

These maskable interrupts are of three varieties, allowing the programmer considerable choice about how interrupts will be treated. The variations are known as Mode Ø, 1 or 2, and the mode can be chosen by a software instruction whose mnemonic is IM (interrupt mode),

taking as its operand the numbers Ø,1 or 2. This instruction will dictate what the response of the Z-80 will be to any subsequent interrupt on the $\overline{INT}$ pin. Whichever mode is selected, the instruction that is in progress at the instant when the interrupt occurs will be completed before the interrupt action starts.

Mode Ø is the default mode which will be used in the absence of any IM instructions, and is selected automatically by a RESET action; it can, of course also be selected in the course of a program by the IMØ instruction. The interrupt signal will normally be passed to the Z-80 by a port, and the port will also be able to provide a byte on the data bus. If the Z-80 PIO is being used, this byte will be taken from the vector address register of the port. This byte will have to be one of the set of RST number bytes which we examined in Chapter 5 and which would also be used with the RST instruction. Thus if a Mode Ø interrupt occurs with the byte 2Ø stored in the vector address register of the port, the effect will be the same as that of a RST2Ø instruction in the program. As you might expect, the programmer will then have to supply a program to deal with the interrupt, located starting at address ØØ2ØH.

A Mode 1 interrupt is a rather less flexible arrangement which causes the Z-80 to load the PC with address ØØ38H and start execution of whatever program is placed at that address. Obviously, a routine to deal with the interrupt must have been written starting at this address with a view to the circumstances which have caused the Mode 1 interrupt. You might, for example, use IM1 in your program just before a cassette operation, and so the routine at ØØ38H would be one specifically intended to deal with any interrupts which occurred during a cassette operation. Another mode of interrupt would then be programmed after the end of this operation. If the interrupt service routine is too long to fit into the space at ØØ38H, then a jump instruction can be placed at this address to lead to another address in the RAM. The contents of the PC before the address ØØ38H is placed in the PC will have been saved on the stack automatically by the interrupt action, and the interrupt routine will have to end with the instruction RETI (return from interrupt), which restores the correct return address in the PC and sets flags to permit another interrupt to take place. The Mode 1 interrupt is useful if the same response to all interrupts is needed, or if a specific interrupt is needed for a specific cause which can be identified and programmed for by using IM1 in the main program. Unlike some other modes, the use of Mode 1 does not require any specific hardware like a port.

The third interrupt mode is a much more flexible system, which provides an interrupt address (or vector) which can be located anywhere in the memory. The contents of the interrupt vector register (I-register) of the Z-80 are used for the upper byte of an address, and the lower byte is provided on the data bus by the port which has caused the interrupt —

once again, the port must contain a register which can store this byte. This interrupt needs more in the way of setting up, because the I-register of the Z-80 will have to be loaded at some stage before the interrupt is used, and the port will also have to be configured so as to place the correct byte in the vector address register. As before, the previous contents of the PC are saved on the stack, and will have to be recovered at the end of the interrupt routine by the use of the RETI command.

The non-maskable interrupt, by contrast, has a much more limited action. As usual, the Z-80 completes the instruction that is currently being executed and clears an interrupt flag in the interrupt register I so as to disable any further interrupts. The address in the PC is then placed on the stack, and the address ØØ66H, called the 'NMI vector' is loaded into the program counter register. Any routine which is going to be used as the response to a non-maskable interrupt must be written starting at ØØ66H, and terminated with RETN (*not* RETI), meaning Return from NMI. The RETN instruction will carry out the restoration of the PC to the address that it had before the interrupt started, and will also reset the NMI interrrupt flag in the interrupt register so that it can thereafter accept another interrupt.

All of these interrupts place the responsibility of saving registers other than the PC firmly on the shoulders of the programmer. Unless registers are saved at the start of the service routine, and restored at the end, the program that was being executed at the start of the interrupt cannot possibly be resumed. In some cases, it might not have to be resumed — for example, if a cassette load were interrupted there would be no point in resuming the load action because the timing would then be incorrect. In such a case there would be no point in saving all the registers, and the interrupt routine would have to ensure that an error message was printed, requiring the user to re-start the load.

8

Techniques

The difference between an experienced assembly language programmer and a beginner is often said to be about a kilobyte. Once you have generated this much code, then you have decidedly shed your 'L' plates and joined the ranks of the experienced. If you have decided that you must master assembly language and you have a lot of problems to solve by its use, this kilobyte may take you only a few weeks. Once the *principles* of assembly language programming have been grasped, attempting to solve problems for yourself will provide experience faster, and of a more useful nature, than any amount of reading could ever provide. That is why this book has concentrated on principles. Since I have set out to cater for the beginner, it seemed pointless to devote a lot of space to examples which will not mean a lot to anyone who is unfamiliar with the hardware of a system for which the examples are written. What I can usefully do is to encourage you to write your own assembly language programs and to show what can be tackled, what ought not to be tackled, and what can be done to check for faults. I'll start with the last problem.

Debugging

A fault in a program is called a bug; finding it and removing it is called debugging; and there's probably a name for the programmer who put the bug in place. If you are using an assembler, then faults of syntax, incorrect jumps, and all of the minor headaches (such as incorrect code) should have been sorted out, so that if a program will not run it will not be because you typed the wrong code somewhere, but because the program plan is defective in some respects. Any attempt to debug the program must start, therefore, with a close look at the program plan, and also at a well-commented copy of the assembly language program. If you didn't draw up a program plan earlier, it's rather late to start now; you'll also find that a printer is a very valuable peripheral, because faults on a printed copy of the assembly language program are very often easier to spot than when they are hand-written on a piece of paper.

Unless the program is a very short one, in which case the fault should

not be difficult to find, the first step is to find at which part of the program things started to go wrong. A linear program (containing no branches or loops) is the simplest to deal with, as all it can do when it goes wrong is to come back with an incorrect number (which might be an ASCII code) in a register. On the rare occasions when a linear program causes the computer to lock up, it must be because there is a piece of incorrect code which the Z-80 has read as a jump instruction to some impossible address. In either case, it should not be difficult to find out what has happened by looking at the number which has been returned and comparing it with the number which should have been returned. Has a SBC been executed in place of a SUB, perhaps, or even an ADD in place of SUB? It may seem obvious, but it's never the easiest thing in the world to find faults in your own creations.

A much more difficult prospect is to find the error in a program that contains a loop, particularly when the loop has been endless when the program was tried. To recover control, you may have to reset the Z-80, clearing all the registers and so wiping out most of the evidence. It's useful to have a special interrupt subroutine for analysing programs, using the MI and writing the subroutine so that register contents are saved on the stack and printed out later, but at the start of your assembly language programming career, it's rather a lot to expect. Even when the registers are cleared by a reset, though, if the program uses RAM it may be possible to tell what has happened by examining the contents of the RAM that has been used. In most cases where an endless loop has occurred, the obvious place to look are the loop-termination conditions — a JR NZ may have been used in place of JR Z, or JR NZ in place of JR NC. Another possibility is that a counter byte is being corrupted, as when DJNZ is being used and some other byte is placed in the B-register on each pass of the loop, preventing the count from decrementing to zero.

Sometimes it is extremely difficult to see a fault in the program plan or in the assembly language. This is usually a case of programmer's myopia, a condition that makes it simple to find faults in other people's programs, but which leaves you totally blind to faults in your own efforts. Authors suffer from similar problems. When a look at the paper-work is not enough, the fault must be located by operating on the program itself.

How you operate on the program depends very much on what assembler is in use, or what monitor can be loaded in place along with the program. Most assemblers and monitors will permit a breakpoint to be inserted, meaning a byte that will cause the Z-80 to return to the operating system, dumping the contents of its registers into a spare piece of RAM as it does so. By placing the breakpoint in the middle of the program and then running the program you can inspect the contents of

registers and memory at the breakpoint position, assuming the program reaches the breakpoint, and so decide if this state of affairs is as you intended it to be. If the program never reaches the breakpoint, then the first half of the program is faulty (look for a loop or jump which might be faulty), and the breakpoint can be inserted half way along the first half. If the first half has gone according to plan, the breakpoint should be inserted half-way along the second half. In either case, the original breakpoint should be replaced by the correct code, or by a NOP instruction. The program can then be run again, and if the break occurs, the registers can be inspected to determine the fault. By continuing in this way, faults can be located remarkably quickly, and if you have some idea about what may be causing the fault, the use of only one breakpoint may be sufficient. If, for example, you have a faulty loop that is not terminated, then, if there is only one loop, the place to put the breakpoint is just before the jump part of the loop, to see what flag settings exist at the instant before the jump is executed.

The use of breakpoints will sort out 99 per cent of the problems that are not obvious from an inspection of the program plan and the assembly language. The stubborn problems remaining will need more drastic action, the best type of which is single stepping.

Single stepping means, as the name suggests, executing the program one command at a time. Single-step operation is a feature of MENTA, but it is not nearly so easy to achieve if an assembler program or monitor is being used along with a computer, since single-step operation is seldom provided. Lucky owners of the TRS-80 can use the remarkable STEP 80 monitor from Mumford Microsystems (USA) which will permit any machine-code routine in ROM or RAM to be executed one step at a time, even when there is interaction with BASIC. STEP 80 allows subroutines to be performed at full speed if required, so that the user can concentrate on single stepping through the main program. At each step, the contents of registers and of any memory involved in the step will be displayed on the screen.

Single stepping is a very powerful method of fault-finding, but it can take a long time to carry out, particularly if the fault occurs only after a number of passes through a loop have been performed. It should be reserved as a last-ditch method for that reason.

Using existing subroutines

When a Z-80 assembly language program is written from scratch, to be used in a machine-control system for example, then every instruction will have to be typed into the assembler. When the assembly language is intended to be assembled into a routine that will run in a Z-80 based

computer, however, it is very often possible to cut corners by making use of routines that already exist in the ROM of the computer. Practically all of the computer subroutines will be written with a RET instruction at some convenient endpoint, so that if they are called using CALL Address, then control will pass back to your own routine after the subroutine has ended. Knowledge of these subroutines can therefore save the programmer a lot of valuable time.

The snag is in getting to know the subroutines. For machines that have been available for some time, such as the TRS-80 and the ZX80/81, disassembled printouts of the ROM routines with comments on how the routines can be used are available. These are generally not available from the manufacturers, who sometimes give the impression of being less than delighted that anyone should wish to know more about their routines, but are available from independent suppliers (see Appendix C). These suppliers have, by making this information available, greatly contributed to the usefulness of a computer. Given the choice, I would opt for a computer system that is fully understood (and for which everything is available!) rather than one which is not, and most machine-code programmers can write much more useful programs when they can understand the operating system of their chosen computer thoroughly. Unless you are very experienced in assembly language, familiar with the principles of design of BASIC interpreters, and also aware of the methods that a given designer uses, you will not necessarily be able to make very much of a disassembled listing of a computer ROM. Even the slightest help in the form of a few addresses can be very helpful, because the effects can be tested by calling these addresses and the results examined. You will not necessarily know what bytes must be placed in which registers, but this can then be discovered from a disassembly once you know where to start.

Subroutine libraries

In the course of writing assembly language programs, you will find that you tend to use the same subroutines over and over again. After some time, you will find that you have a useful subroutine library which you can call upon when writing assembly language programs, but it does not follow that you will find this library of subroutines useful unless they have been documented in a form which is convenient. Good documentation should include details of what registers are used in each subroutine, what values must exist in each register before a subroutine is called, whether the subroutine is relocatable, how quickly it will run, whether it can be interrupted without damage, and what values will be present in the registers or in RAM after it has run. The magazine

Personal Computer World runs a column called SUBSET which publishes subroutines documented in a way that should be used as a model by every Z-80 programmer. As it happens, most of the entries in SUBSET are for the Z-80, reflecting the number of serious machine-code programmers who write for this type of microprocessor.

A few assemblers permit the use of 'macros'. A macro is a piece of assembly language which can be required at more than one place in a program and which can be assembled simply by naming it. The ability to use macros allows the user to create assembly language programs much more rapidly than is the case when each item must be typed in. An assembler with macro capabilities along with a set of library subroutines is an excellent collection of support software for the assembly language writer. Do not, incidentally, confuse a macro with a subroutine. A subroutine is a piece of code that is placed in one part of a program and used by CALLs from various places in the program. The macro is a program-writing device that causes the assembler to place identical pieces of code in several different parts of the same program without the effort of typing the whole code each time.

Interfacing BASIC with machine code

A problem that can be solved using BASIC can always be solved also by the use of machine code, though a problem that can be solved by the use of machine code may very well *not* be solvable by the use of BASIC. It does not follow, however, that every programming problem *should* be solved by the use of machine code. The problems of handling mathematical work with machine code are so great that only a dedicated card-carrying masochist would want to use machine code for this purpose if there is any way of avoiding the effort. Writing mathematical routines, particularly the more elementary arithmetic routines, is akin to re-inventing the wheel — the best routines have all been written many times before. Wherever your program uses any calculation other than a simple addition or subtraction that can be tackled directly by the use of ADD or SUB, or counting, then BASIC should be used. If you are assembling code for an application which does not include the presence of a computer, then use the arithmetical routines from the ROM of a Z-80 based computer. Life is much too short to spend it on devising new arithmetical or mathematical routines.

At the other extreme, a fast-moving but simple action game is a problem asking for a machine-code solution. Very few Z-80 based machines have a version of BASIC that runs fast enough for such games, forcing the programmer either to use a more suitable language, such as FORTH, or to program directly in machine code.

Many programs, however, need a mixture of both high and low-level languages. Many of the sections of a program, such as instructions, screen prompts, in-outs, printouts, etc. can be done in BASIC, along with calculations, and only the time-critical actions such as graphics animation and string sorting need to be done using machine code. Another combination is a straightforward BASIC program which requires some form of input or output, such as input through the cassette port from another machine, that is non-standard. Such an application needs a machine-code subroutine called from a program which is otherwise in BASIC.

Mixtures of BASIC and machine code

Most computers provide for calling a machine-code routine from BASIC by means of the USR BASIC command. The last step in such a process must consist of loading the program counter of the Z-80 with the starting address of the machine-code routine, and two distinct methods of doing this are used. One is to load the starting address from BASIC into two memory addresses that are reserved for this purpose. When the USR routine is called from BASIC, it obtains this starting address for the machine code from these memory locations, and places this address into the PC. A USR call can then take an argument, meaning a number that follows the USR command, such as USR(Ø), USR(24Ø), and so on. This number will then be passed to the machine code routine by being placed in a register or in a register pair when the machine-code section starts. This is all done by the USR command; no attention to this is needed in the routine itself. The alternative method is to follow USR by the address at which the routine starts — a simpler scheme, but one which makes no direct provision for passing any other numbers into the routine. The first method is used by the TRS-80, the second by the Spectrum, to name two well-known examples.

These methods are not necessarily the most convenient. Sometimes it is desirable that the programmer should be able to call the subroutine by pressing a key or by typing a word. Where the operating system of the computer allows, this is a fairly simple scheme to carry out, and has the advantage that it can be used to provide extensions to the BASIC language of the computer. To use this method, however, the operating system must contain RAM exits. These are addresses to which the operating system jumps during execution of keyboard actions or during the execution of some BASIC commands. Normally these addresses will be loaded with bytes which cause a jump back to the operating system in the ROM, but since these instructions are in RAM, they can be changed. The purpose of such RAM calls is usually to allow a disc

operating program to be added into the RAM, so that most if not all of these useful addresses may be used when a disc system is added. Others, however, can be used to intercept calls and to patch in your own routines. Just one of these addresses will be sufficient, because as many of your own routines as you like can be chained from it as long as there is a suitable return address at the end of the routines so that normal operation can be resumed. This would be the address which was held in the RAM space originally.

Not all computers are equipped with these useful exits, however, and owners of TRS-80s or of the promising new LYNX from Camputers of Cambridge have every reason to feel smug about the facilities that they can add so easily to their machines. In many cases, however, intensive beavering by a user group will reveal exits of this type on computers whose manufacturers refuse to publish any information about these useful addresses.

Some care has to be taken over returning to the operating system. When your routine has been called by USR in BASIC, then the correct return is usually ensured by a RET at the end of your program, but for some other types of call, a JP to an address in the operating system will be needed. This is something that varies from one type of machine to another, and the manual will have to be consulted — though the user group may know considerably more than the manual reveals! When your machine-code routine has been called by using an exit address the end of the routine must contain a jump back to the operating system, as we noted earlier.

Passing variables

When a machine-code routine is used along with BASIC, a problem that frequently arises is that of passing variables from the BASIC program to the machine code and back again. For example, if a machine-code routine is used for graphics movements, the BASIC program will need to provide the ASCII code number of the graphics character or the name of a string variable which holds graphics characters, and the starting position, finishing position and speed of movement. If a string-sorting program uses a machine-code sort, then each item in the string array will have to be available to the machine code, meaning that the length and address of each string must be in memory addresses that the machine code can make use of. There are two main methods for passing variables to the machine-code routine when no provision for this has been made in the design of the BASIC USR call. One method is to POKE values from the BASIC program into a piece of reserved memory, and to use the same memory addresses in the machine-code routine. This is a

```
        1000 AH = INT (A/256) : AL = A – 256 * AH
BASIC   1010 POKE 32767, AH : POKE 32766, AL
                        .
                        .
                        .
                        .
                        .
              LD HL, (32766) ; two bytes into HL
```

Fig. 8.1. Passing an integer variable value to a machine-code routine. The variable A is split into two bytes by BASIC line 1000, and poked into memory in line 1010. This allows the machine-code program to load the bytes into a double register as shown

reasonable method when only a few values have to be passed, and it has the merit of being simple to use, as Fig. 8.1 demonstrates. The other method is to access the variable names and values directly from the computer. All computers create a 'variable list table' in the course of a BASIC program, and the starting address of this variable list table will be held somewhere in the RAM. The location of the table is generally immediately above the BASIC program space, and the reason for keeping its starting address in RAM is that this space is 'dynamically allocated', meaning that it will shift addresses each time lines are added to, or deleted from, the BASIC program.

To make use of the variable list table (VLT) you have to know the structure of the table for your own machine, and not all manufacturers are forthcoming about the form of entries. The Spectrum manual is good in this respect, the NASCOM manual has the information (if you can find it) and some others are equally good, but in some cases the user group is a better bet for information. Once the structure of the VLT is known, a machine-code routine can be written to access the variables that are needed. The usual form of this routine will be to have the starting address of the VLT in the HL register pair, and to step through the VLT, using INC HL in a loop, loading A from (HL) until the correct byte appears. The form of the byte that has to be recognised will be decided by the way the computer stored its variables — you will probably have to carry out a recognition step on the first byte in the VLT, and if this is not the one you want, step several bytes to the next one. If this is several bytes (seven, for example) it is easier to step by using an ADD step rather than by a number of INC instructions.

The form of program that you need depends on how the computer stores the VLT bytes. One common method that is used for strings is to store a 'key' block consisting of three bytes, for each string, rather than storing the string itself in the VLT. The key block then consists of one byte, which is the number of characters in the string, followed by the two bytes which provide the starting address for the string. In this way, the entries in the VLT are all of three bytes, regularly spaced (plus bytes for the name of the string), rather than the irregular spacing that would be

used if the strings themselves were kept in the VLT. String arrays are generally stored in order, so that once the first item of a string has been found, the rest of them can be located easily by counting bytes from the first item.

Calling BASIC from machine code

The usual method of using machine code along with BASIC is to call machine-code from the BASIC, but it may be equally necessary to call BASIC from the machine code. This is a very much more difficult task unless the structure of the BASIC is well understood, because though it is possible to jump from a machine code program to the start of a BASIC line, this does *not* cause that BASIC line to be executed. What is needed is to place the 'token' (the one-byte instruction) that is used for RUN into the correct register, and call the correct subroutine. An alternative is to use the GOTO token and subroutine, but how these are found and in what register the token must be placed are matters for investigation on your own computer. Once again, well-established systems tend to be the best documented, because someone, somewhere, will have done this and written about it. Alternatively, there will be clues in a detailed disassembly of the operating system which will tell you what you need to know. A lot of care is needed to set up the operating conditions correctly, however.

Storage space

As we draw to the end of this guide, some comments on the storage of machine-code routines in the RAM are needed. One method that is used on many computers is to use a CLEAR command from BASIC to reserve memory at the top end of the available address range. This can often be done just as easily in machine code. Another method that was used in early TRS-80 days was to store the bytes of a machine-code program as a number array or as a long string — this is possible because the TRS-80 has the useful VARPTR instruction which will find the address for any variable name. A method pioneered by ZX-81 programmers is to store bytes following a REM in a BASIC line, allowing enough space when the REM line is written for as many bytes as will be used. The REM line can be filled with padding characters or spaces — the advantage of using padding characters is that they can be more easily counted.

One method that is less frequently used is to shift BASIC itself. Most computers keep the 'start of BASIC' address stored in two bytes of RAM, and by altering this address, the BASIC program can be stored at another place in memory. This is not — indeed cannot — be done when the BASIC program is already in place, of course, but before the BASIC is loaded from tape or disc or by typing. This leaves a space in the lower RAM addresses below the start of the BASIC address, which can be used for machine-code programs. This space will be protected against use by any action of the machine operating system, because it is 'no-mans land', below BASIC and above the RAM that is used for storing addresses. Usually only the 'start of BASIC' address has to be changed, but in a few cases, the VLT address may have to be altered as well. This action is usually automatic, however, as it is done by the machine operating system when the new address appears in the 'start of BASIC'.

The next step

This book has been concerned primarily with methods, much less with examples. The trouble with examples is that they usually relate to specific problems, in which case they are of little use to the reader who has a different sort of problem, or they are so general that they are almost meaningless. Rather than scatter examples throughout the book, then, I have illustrated the use of assembly language programming with one fairly massive example in Appendix A. This is a fully commented example of the program which I wrote for the TRS-80. It enables any data, text or graphics in place on the screen to be dumped on to a cassette or to a printer, and similarly allows a cassette to be replayed to the screen. I have chosen this example because it shows the interactions between the machine-code program and the operating system of the machine rather well, and also how the routines in the ROM have been used, along with the RAM exits for the (non-disc) system of the 16K TRS-80.

The next step for you, the reader, is to extend your experience of programming in assembly language. Two useful books for this purpose are *The Z-80 Instruction Set,* published by SGS-ATES, which describes each type of instruction in detail, and Lance Leventhal's classic *Z-80 Assembly Language Programming* (Osborne, now McGraw-Hill, Berkeley, Ca.) which is an encyclopaedia of methods for the programmer who has already cut his assembly language teeth. For the more advanced and more ambitious, a superb book is Brown's *Writing Interactive Interpreters and Compilers* (Wiley, Chichester) which, in spite of its forbidding title, contains a wealth of advice on good programming methods and is full of useful information. Another text, suited for the more experienced is

Essential Software Tools, by Kernighan and Plauger (Addison-Wesley, New York). In addition the SUBSET series in *Personal Computer World* is a constant source of useful routines and ideas. The most important learning aid, however, is to pit your own wits against your own problems. Good luck!

Machine-specific books

TRS-80

Microsoft BASIC Decoded, by James Farvour, is a complete labelled disassembly of the BASIC routines of the TRS-80. The coding is given in general form to avoid copyright legal problems, but in conjunction with the printout from a disassembler, this constitutes 12K of Z-80 code with very detailed explanations.

BASIC Faster and Better, by Lewis Rosenfelder, is a book of subroutines, many of them in POKEd machine code, with assembly language versions, which will enable the TRS-80 to outperform many more modern designs. The book is particularly useful for its methodical approach, and its detailed descriptions of how to interleave machine code with BASIC.

ZX81

Understanding Your ZX81 ROM by Dr Ian Logan. A list of Z-80 instructions with details of how these instructions are used in the ROM of the ZX-81.

Sinclair ZX81 ROM Dissassembly Part A and *Part B,* Part A by Dr Ian Logan, Part B by Dr Ian Logan and Dr Frank O'Hara. This is not just a dissassembly with a few comments, but a breakdown with a lot of detail of each routine, taken in order, with a guide as to how the routines fit together. It is undoubtedly a very useful guide to the ZX-81 routines, and contains much information that will be useful to the Z-80 assembly language programmer. The mathematical routines in Part B, for example, will be very useful to anyone who needs to use such routines in a program that is not part of an existing computer.

Appendix A

Sample program listing

This is an example of a program which was developed to provide the TRS-80 Model 1, non-disc machine with the facilities to dump the contents of the screen to tape or to the printer, and to load back from the cassette to the screen memory. The program is in several quite separate sections. Lines 7 and 8 deal with the autostart, placing a byte in the reserved RAM of the machine at an address which will cause the program to start automatically. This can be done on many machines, but the method shown here is specific to the TRS-80. Lines 9-14 then define addresses that will be used in the assembly language. Lines 15-24 set the starting addresses of the three main sections of the program into the RAM call addresses that are used when the words PUT, GET and SAVE occur in a BASIC program. This enables each section of the machine-code program to be called independently from within a BASIC program by simply using these reserved words, which are unused in Level 2 BASIC. At lines 22 and 23, the autostart is cancelled by replacing the original byte into memory — this ensures that subsequent programs do not autostart by default. Lines 25 and 26 then cause a jump back to BASIC with a space cleared in memory for the machine code. This space includes 1026 bytes of buffer space which is used to hold bytes temporarily while awaiting the tape dump.

At the end of running the initialising section the computer will be back in BASIC, and the other sections of machine code will run only when called by the appropriate reserved words. The tape dump section starts at line 28, using the alternate registers so as to avoid corrupting the registers used by BASIC. This makes use of the fact that the TRS-80 Level 2 BASIC does not make any use of the alternate registers.

The scheme of storing the screen bytes is not simply one of storing each byte; it has been designed to omit spaces, so that it is much faster for the type of program for which it was designed — one containing graphics and text, but with many spaces. The scheme is to scan each byte in the screen memory space. If the byte is a space, it is ignored, but when a non-space character is found its address is placed in the buffer memory, followed by the byte itself. If the following bytes are also not blanks, they are placed into the buffer in sequence. When the next space is found, a

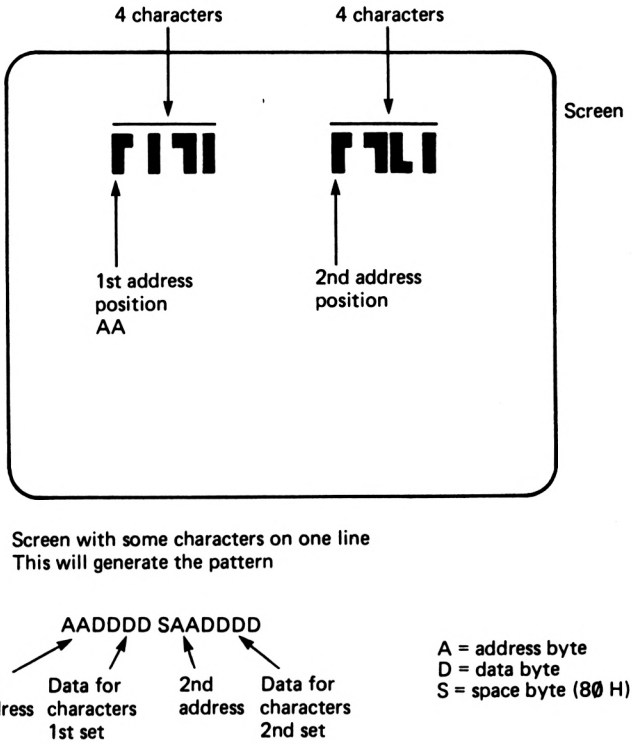


Fig. A.1. How the data stored in memory is formatted

8ØH character (which on the TRS-80 is a blank which cannot be obtained from the keyboard) is placed in the buffer, but the blanks in the screen memory are once again skipped. In this way, large areas of blank space are simply not recorded — the format of the data stored in memory is shown in Fig. A1. In the course of filling the buffer memory, a flag byte is set when a space is found and this is used as the signal to insert the 8ØH byte into the buffer — the flag is reset when a character is found. At the start of the program, the highest possible screen address has been stored in BC, and on each read of a byte, this address is compared with the actual address so as to determine when to terminate the loading of the buffer. Note that the program as shown does not use the whole of the screen memory — the top and bottom lines were not included so that they could be used for other purposes. When the buffer is filled with as many bytes as constitute a record of the screen (usually considerably less than the 1026 allowed), the count is then transferred into BC, with the buffer starting address in HL. The count bytes are recorded on tape,

followed by the contents of the buffer. At the end of the recording the registers are restored and the program returns to BASIC.

When a load from cassette is called, the registers are exchanged again, and the first two bytes are read from the tape. These are the count bytes, and are placed into BC, which is then decremented twice to allow for these two bytes. The other bytes are then read in turn, using the 8ØH indicator byte to determine whether what follows is two bytes of an address or a byte of data. The addresses are used to locate the position in the screen memory of the next data byte, and if several bytes follow, the screen memory is incremented after each byte has been put in place. Only as many load operations as there are bytes of data are performed to the screen memory. When all of the bytes have been read and disposed of, so that BC is decremented to zero, the registers are restored and the program returns to BASIC.

In the printing section, the screen is divided up into its 64 characters per line and 15 lines (one left for screen messages), and characters are loaded from the screen addresses in turn to the printer, sending a carriage return to the printer at the end of each line. This is not quite so straightforward as our earlier simple example, in which we assumed that the screen layout corresponded exactly with the printer format. As usual, the normal register set is restored at the end of the section.

Note the example of programmer's myopia in lines 153, 154, 155 and again in 161, 162, 163 where the procedure for DEC BC is followed needlessly out of force of habit! In each case, the DEC step could simply have been followed by the jump step, because decrementing a single register will set flags. I have no excuses to offer!

Program listing

```
1;
2;
3;
4;PROGRAM TO DUMP SCREEN CONTENTS
5;TO TAPE, AND TO LOAD BACK FROM
6;TAPE. SCREEN SPACES ARE IGNORED
7 ORG 41E2H ; FOR AUTOSTART
8 JP(HL) ; PUT AUTOSTART BYTE IN
9 ORG 7AF8H ; START OF MAIN PROGRAM
10 VIDEO : EQU 3C4ØH ; START OF SCREEN MEMORY
11 TOPVID: EQU 3FBFH ; TOP OF SCREEN MEMORY
12 PUT : EQU 4183H ; ADDRESS OF RAM CALL
13 GET : EQU 418ØH ; RAM CALL
```

```
14 SAVE : EQU 41A1H ; RAM CALL
15 EXX ; PRESERVE NORMAL REGISTERS
16 LD HL, STARTIT ; ADDRESS OF DUMP START
17 LD(PUT),HL ; INTO PUT ADDRESS
18 LD HL, BEGIN ; ADDRESS OF LOAD START
19 LD(GET),HL ; INTO GET ADDRESS
20 LD HL, PRNTR ; ADDRESS OF PRINT START
21 LD(SAVE),HL ; INTO SAVE ADDRESS
22 LD HL,41E2H ; CANCEL AUTOSTART BY
23 LD(HL),ØC9H ; RESTORING ORIGINAL BYTE
24 EXX ; NORMAL REGISTERS BACK
25 LD HL, STARTIT - 2 ; PROTECT MEMORY
26 JP ØØE7H ; BACK TO BASIC
27 STARTIT : EX AF,AF ; ALTERNATE SET
28 EXX ; TO PRESERVE BASIC
29 LD HL, VIDEO ; START OF SCREEN
30 LD DE, TEMP + 2 ; BUFFER SPACE
31 LD BC, TOPVID ; TOP OF SCREEN MEMORY
32 FIRST: LD A,(HL) ; FIRST BYTE
33 CP 2ØH ; IS IT SPACE?
34 JR NZ,RECADR ; WRITE ADDRESS IF NOT
35 GETIT: LD A,(HL) ; SAME BYTE
36 CP 2ØH ; IS IT SPACE?
37 JR Z,FLAG ; FLAG IT IF IT IS
38 CP 8ØH ; OTHER BLANK CHARACTER
39 JR Z,FLAG ; SO FLAG IT
40 PUSH AF ; SAVE THIS BYTE
41 LD A,(STOR) ; TEST FLAG BYTE
42 CP Ø1H ; IS FLAG SET?
43 JR Z,PUTSPC ; DUMP A SPACE
44 POP AF ; GET BYTE BACK
45 PUTBYT: LD (DE),A; STORE IN BUFFER
46 INC DE ; BUMP UP BUFFER ADDRESS
47 BUMP: INC HL ; BUMP UP VIDEO
48 PUSH HL ; SAVE IT
49 OR A ; TO CLEAR CARRY
50 SBC HL,BC ; CHECK FOR END
51 POP HL ; RESTORE ADDRESS
52 JR Z, DUMP ; OUT IF END
53 JR GETIT ; BACK FOR ANOTHER
54 FLAG : PUSH AF ; SAVE BYTE
55 LD A,Ø1H ; FLAG SET BYTE
56 LD (STOR),A; PUT IT IN MEMORY
57 POP AF ; RESTORE BYTE
```

```
58 JR BUMP ; BACK AGAIN
59 PUTSPC: LD A, 8ØH ; SPACE BYTE
60 LD (DE),A ; PUT INTO BUFFER
61 INC DE ; BUMP UP BUFFER
62 XOR A ; CLEAR A
63 LD (STOR),A; CLEAR FLAG
64 POP AF ; RETURN BYTE
65 RECADR: PUSH AF ; SAVE BYTE
66 LD A,L ; LOWBYTE OF
67 LD (DE),A ; ADDRESS AND
68 INC DE ; THEN . . .
69 LD A,H ; THE HIGH BYTE
70 LD (DE),A; INTO DE
71 INC DE ; BUMP BUFFER
72 POP AF ; RESTORE BYTE
73 JR PUTBYT; ANOTHER BYTE
74 STOR: DS 1 ; RESERVE PLACE FOR FLAG BYTE
75 DUMP : EX DE,HL ; STORE TOP MEM. IN HL
76 LD DE, TEMP ; BUFFER START IN DE
77 OR A ; CLEAR CARRY
78 SBC HL,DE ; FIND HOW MANY BYTES
79 LD B,H ; NEED ANSWER . . .
80 LD C,L ; INTO BC
81 LD HL, TEMP ; START OF BUFFER NOW IN HL
82 LD (HL),C ; COUNT LOW AT START OF BUFFER
83 INC HL ; BUMP BUFFER ADDRESS
84 LD (HL),B ; COUNT HIGH IN BUFFER
85 DEC HL ; BACK TO START OF BUFFER
86 LD A,ØØH ; CLEAR A SO AS TO . . .
87 CALL Ø212H ; SELECT TAPE NO. 1
88 PUSH BC ; USED BY CALL
89 CALL Ø287H ; START TAPE MOVING
90 POP BC ; RESTORE COUNT
91 WRITE: LD A,(HL) ; FIRST BYTE TO A
92 CALL Ø264H ; RECORD BYTE ON TAPE
93 INC HL ; BUMP BUFFER
94 DEC BC ; DECREMENT COUNT
95 LD A,B ; CHECK END
96 OR C; OF COUNT
97 JR NZ, WRITE ; AGAIN IF NOT ENDED
98 CALL Ø1F8H ; STOP TAPE IF DONE
99 EXX ; RESTORE NORMAL REGISTERS
100 EX AF,AF ; AND AF TOO
101 RET ; BACK TO BASIC
```

```
102 BEGIN : EX AF,AF ; SAVE REGISTERS
103 EXX ; ALL OF THEM!
104 LD A,ØØH ; CLEAR A SO AS TO
105 CALL Ø212H ; SELECT DRIVE NO. 1
106 LD HL, VIDEO ; SCREEN ADDRESS IN HL
107 CALL Ø296H ; FIND LEADER AND SYNC ON TAPE
108 CALL Ø235H ; READ A BYTE (COUNT LOW)
109 LD C,A ; INTO C
110 CALL Ø235H ; READ HIGH BYTE
111 LD B,A ; INTO B, COUNT NOW IN BC
112 DEC BC
113 DEC BC ; TWO LESS TO READ
114 READ: CALL Ø235H ; READ A BYTE
115 PUSH AF ; SAVE IT ON STACK
116 DEC BC ; DECREMENT COUNT
117 LD A,B ; TO CHECK . . .
118 OR C ; FOR ZERO COUNT
119 JR Z, PUTOUT ; RESTORE AND END IF ZERO
120 POP AF ; RESTORE BYTE IN A
121 CP 8ØH ; IS IT SPACE MARKER?
122 JR Z, TESTAD ; ADDRESS FOLLOWS IF IT IS
123 LD(HL),A ; NOT SPACE, PUT IT ON SCREEN
124 INC HL ; BUMP SCREEN ADDRESS
125 JR READ ; GET ANOTHER ONE
126 TESTAD : CALL Ø235H ; LOW ADDRESS BYTE
127 DEC BC ; COUNT IT
128 LD L,A ; GET IT . . .
129 LD A,B ; CHECK COUNT
130 OR C ; SET FLAGS
131 JR Z,ENDIT ; OUT IF COUNT ZERO
132 CALL Ø235H ; GET HIGH ADDRESS BYTE FROM TAPE
133 DEC BC ; COUNT IT
134 LD H,A ; ADDRESS NOW IN HL
135 LD A,B ; CHECK AGAIN
136 OR C ; FOR COUNT END
137 JR Z,ENDIT ; OUT IF ZERO
138 JR READ ; OTHERWISE BACK FOR ANOTHER
139 PUTOUT: POP AF ; RESTORE AF
140 LD (HL),A; LAST BYTE
141 ENDIT: CALL Ø1F8H ; STOP TAPE
142 EXX ; BACK TO NORMAL . . .
143 EX AF,AF ; REGISTERS AGAIN
144 RET ; BACK TO BASIC
145 TEMP: DS 1Ø26; SPACE FOR BUFFER
```

```
146 PRNTR: EX AF,AF ; SAVE REGISTERS
147 EXX ; BY USING ALTERNATES
148 LD HL,3CØØH ; VIDEO START
149 LD B,ØFH ; LINE COUNT
150 LD C, 4ØH ; CHARACTER COUNT PER LINE
151 LOOP: LD A,(HL); GET BYTE FROM SCREEN
152 CALL ØØ3BH ; SEND IT TO PRINTER
153 DEC C ; COUNT IT
154 LD A,C ; TEST IT
155 OR A ; SET FLAGS
156 JR Z,NEWLIN; NEWLINE IF ZERO
157 INC HL ; NEXT ADDRESS
158 JR LOOP ; BACK FOR NEXT
159 NEWLIN: LD A,ØDH ; C/R FOR PRINTER
160 CALL ØØ3BH ; SEND IT
161 DEC B ; ONE LINE LESS
162 LD A,B ; LINE COUNT
163 OR A ; SET FLAGS
164 JR Z,RTN ; OUT IF FINISHED
165 LD C,4ØH ; COUNT FOR NEXT LINE
166 INC HL ; NEXT SCREEN ADDRESS
167 JR LOOP ; BACK AGAIN
168 EX AF,AF ; EXCHANGE
169 EXX ; REGISTERS AGAIN
170 RET ; BACK TO BASIC
171 END
```

Appendix B

Z-80 Mnemonics and codes

The following list of Z-80 mnemonics and codes is believed to be complete — though in a list of so many items, it is always possible to omit one or two. Each variation of opcode is shown, so that the user does not have the tedious task of calculating the operand from a list of binary coded arrangements. Each code has been shown both in hex (second column) and in denary (third column), but the operands have been altered from the conventional N, NN, etc. where a number has to be filled in by the user. Where this is needed, a set of zeros (Ø) has been shown, generally two for a data byte and four for an address in the hex version, though the denary version shows only a single zero. The same method has been used in the mnemonic, so that ADD A,(IX + ØØ) means that a displacement byte is required in the code where the zeros occur. This method of indicating bytes supplied by the programmer has been partly forced on me by the computer program that produced the listing, but seems to be reasonably easy to interpret in comparison to the methods that have formerly been used.

ADC A,(HL)	8E	142
ADC A,(IX+00)	DD	221
	8E	142
	00	0
ADC A,(IY+00)	FD	253
	8E	142
	00	0
ADC A,A	8F	143
ADC A,B	88	136
ADC A,C	89	137
ADC A,D	8A	138
ADC A,00	CE	206
	00	0
ADC A,E	8B	139
ADC A,H	8C	140
ADC A,L	8D	141
ADC HL,BC	ED	237
	4A	74
ADC HL,DE	ED	237
	5A	90
ADC HL,HL	ED	237
	6A	106
ADC HL,SP	ED	237
	7A	122
ADD A,(HL)	86	134
ADD A,(IX+00)	DD	221
	86	134
	00	0
ADD A,(IY+00)	FD	253
	86	134
	00	0
ADD A,A	87	135

ADD A,B	80	128
ADD A,C	81	129
ADD A,D	82	130
ADD A,00	C6	198
	00	0
ADD A,E	83	131
ADD A,H	84	132
ADD A,L	85	133
ADD HL,BC	09	9
ADD HL,DE	19	25
ADD HL,HL	29	41
ADD HL,SP	39	57
ADD IX,BC	DD	221
	09	9
ADD IX,DE	DD	221
	19	25
ADD IX,IX	DD	221
	29	41
ADD IX,SP	DD	221
	39	57
ADD IY,BC	FD	253
	09	9
ADD IY,DE	FD	253
	19	25
ADD IY,IY	FD	253
	29	41
ADD IY,SP	FD	253
	39	57
AND (HL)	A6	166
AND (IX+00)	DD	221
	A6	166
	00	0

AND (IY+00)	FD	253
	A6	166
	00	0
AND A	A7	167
AND B	A0	160
AND C	A1	161
AND D	A2	162
AND 00	E6	230
	00	0
AND E	A3	163
AND H	A4	164
AND L	A5	165
BIT 0,(HL)	CB	203
	46	70
BIT 0,(IX+00)	DD	221
	CB	203
	00	0
	46	70
BIT 0,(IY+00)	FD	253
	CB	203
	00	0
	46	70
BIT 0,A	CB	203
	47	71
BIT 0,B	CB	203
	40	64
BIT 0,C	CB	203
	41	65
BIT 0,D	CB	203
	42	66
BIT 0,E	CB	203
	43	67
BIT 0,H	CB	203
	44	68
BIT 0,L	CB	203
	45	69
BIT 1,(HL)	CB	203
	4E	78
BIT 1,(IX+00)	DD	221
	CB	203
	00	0
	4E	78
BIT 1,(IY+00)	FD	253
	CB	203
	00	0
	4E	78
BIT 1,A	CB	203
	4F	79
BIT 1,B	CB	203
	48	72
BIT 1,C	CB	203
	49	73
BIT 1,D	CB	203
	4A	74
BIT 1,E	CB	203
	4B	75
BIT 1,H	CB	203
	4C	76
BIT 1,L	CB	203
	4D	77
BIT 2,(HL)	CB	203
	56	86
BIT 2,(IX+00)	DD	221
	CB	203
	00	0
	56	86
BIT 2,(IY+00)	FD	253
	CB	203
	00	0
	56	86
BIT 2,A	CB	203
	57	87
BIT 2,B	CB	203
	50	80
BIT 2,C	CB	203
	51	81
BIT 2,D	CB	203
	52	82
BIT 2,E	CB	203
	53	83
BIT 2,H	CB	203
	54	84
BIT 2,L	CB	203
	55	85
BIT 3,(HL)	CB	203
	5E	94
BIT 3,(IX+00)	DD	221
	CB	203
	00	0
	5E	94
BIT 3,(IY+00)	FD	253
	CB	203
	00	0
	5E	94
BIT 3,A	CB	203
	5F	95
BIT 3,B	CB	203
	58	88
BIT 3,C	CB	203
	59	89
BIT 3,D	CB	203
	5A	90
BIT 3,E	CB	203
	5B	91
BIT 3,H	CB	203
	5C	92
BIT 3,L	CB	203
	5D	93
BIT 4,(HL)	CB	203
	66	102
BIT 4,(IX+00)	DD	221
	CB	203
	00	0
	66	102
BIT 4,(IY+00)	FD	253
	CB	203
	00	0
	66	102
BIT 4,A	CB	203
	67	103
BIT 4,B	CB	203
	60	96
BIT 4,C	CB	203
	61	97
BIT 4,D	CB	203
	62	98
BIT 4,E	CB	203
	63	99
BIT 4,H	CB	203
	64	100
BIT 4,L	CB	203
	65	101
BIT 5,(HL)	CB	203
	6E	110
BIT 5,(IX+00)	DD	221
	CB	203
	00	0
	6E	110
BIT 5,(IY+00)	FD	253
	CB	203
	00	0
	6E	110
BIT 5,A	CB	203
	6F	111
BIT 5,B	CB	203
	68	104
BIT 5,C	CB	203
	69	105
BIT 5,D	CB	203
	6A	106
BIT 5,E	CB	203
	6B	107
BIT 5,H	CB	203
	6C	108
BIT 5,L	CB	203
	6D	109
BIT 6,(HL)	CB	203
	76	118
BIT 6,(IX+00)	DD	221
	CB	203
	00	0
	76	118

BIT 6,(IY+00)	FD	253
	CB	203
	00	0
	76	118
BIT 6,A	CB	203
	77	119
BIT 6,B	CB	203
	70	112
BIT 6,C	CB	203
	71	113
BIT 6,D	CB	203
	72	114
BIT 6,E	CB	203
	73	115
BIT 6,H	CB	203
	74	116
BIT 6,L	CB	203
	75	117
BIT 7,(HL)	CB	203
	7E	126
BIT 7,(IX+00)	DD	221
	CB	203
	00	0
	7E	126
BIT 7,(IY+00)	FD	253
	CB	203
	00	0
	7E	126
BIT 7,A	CB	203
	7F	127
BIT 7,B	CB	203
	78	120
BIT 7,C	CB	203
	79	121
BIT 7,D	CB	203
	7A	122
BIT 7,E	CB	203
	7B	123
BIT 7,H	CB	203
	7C	124
BIT 7,L	CB	203
	7D	125
CALL 00	CD	205
	00	0
CALL C,00	DC	220
	00	0
CALL M,00	FC	252
	00	0
CALL NC,00	D4	212
	00	0
CALL P,00	F4	244
	00	0
CALL PE,00	EC	236
	00	0
CALL PO,00	E4	228
	00	0
CALL Z,00	CC	204
	00	0
CCF	3F	63
CP(HL)	BE	190
CP(IX+0)	DD	221
	BE	190
	00	0
CP(IY+0)	FD	253
	BE	190
	00	0
CP A	BF	191
CP B	B8	184
CP C	B9	185
CP D	BA	186
CP 00	FE	254
	00	0
CP E	BB	187
CP H	BC	188
CP L	BD	189
CPD	ED	237
	A9	169
CPDR	ED	237
	B9	185
CPI	ED	237
	A1	161

CPIR	ED	237
	B1	177
CPL	2F	47
DAA	27	39
DEC(HL)	35	53
DEC(IX+0)	DD	221
	35	53
	00	0
DEC(IY+0)	FD	253
	35	53
	00	0
DEC A	3D	61
DEC B	05	5
DEC BC	0B	11
DEC C	0D	13
DEC D	15	21
DEC DE	1B	27
DEC E	1D	29
DEC H	25	37
DEC HL	2B	43
DEC IX	DD	221
	2B	43
DEC IY	FD	253
	2B	43
DEC L	2D	45
DEC SP	3B	59
DI	F3	243
DJNZ,00	10	16
	00	0
EI	FB	251
EX(SP),HL	E3	227
EX(SP)IX	DD	221
	E3	227
EX(SP)IY	FD	253
	E3	227
EX AF,AF'	08	8
EX DE,HL	EB	235
EXX	D9	217
HLT	76	118
IM 0	ED	237
	46	70
IM 1	ED	237
	56	86
IM 2	ED	237
	5E	94
IN A,(C)	ED	237
	78	120
IN A,PORT	DB	219
	00	0
IN B,(C)	ED	237
	40	64
IN C,(C)	ED	237
	48	72
IN D,(C)	ED	237
	50	80
IN E,(C)	ED	237
	58	88
IN H,(C)	ED	237
	60	96
IN L,(C)	ED	237
	68	104
INC(HL)	34	52
INC(IX+00)	DD	221
	34	52
	00	0
INC(IY+00)	FD	253
	34	52
	00	0
INC A	3C	60
INC B	04	4
INC BC	03	3
INC C	0C	12
INC D	14	20
INC DE	13	19
INC E	1C	28
INC H	24	36
INC HL	23	35
INC IX	DD	221
	23	35
INC IY	FD	253
	23	35

```
INC L          2C  44
INC SP         33  51
IND            ED  237
               AA  170
INDR           ED  237
               BA  186
INI            ED  237
               A2  162
INIR           ED  237
               B2  178
JP(HL)         E9  233
JP(IX)         DD  221
               E9  233
JP(IY)         FD  253
               E9  233
JP 0000        C3  195
               00  0
               00  0
JP C,0000      DA  218
               00  0
               00  0
JP M,0000      FA  250
               00  0
               00  0
JP NC,0000     D2  210
               00  0
               00  0
JP NZ,0000     C2  194
               00  0
               00  0
JP P,0000      F2  242
               00  0
               00  0
JP PE,0000     EA  234
               00  0
               00  0
JP PO,0000     E2  226
               00  0
               00  0
JP Z,0000      CA  202
               00  0
               00  0
JR C,00        38  56
               00  0
JR 00          18  24
               00  0
JR NC,00       30  48
               00  0
JR NZ,00       20  32
               00  0
JR Z,00        28  40
               00  0
LD(0000),A     32  50
               00  0
               00  0
LD(0000),BC    ED  237
               43  67
               00  0
               00  0
LD(0000),DE    ED  237
               53  83
               00  0
               00  0
LD(0000),HL    ED  237
               63  99
               00  0
               00  0
LD(0000),HL    22  34
LD(0000),IX    DD  221
               22  34
               00  0
               00  0
LD(0000),IY    FD  253
               22  34
               00  0
               00  0
LD(0000),SP    ED  237
               73  115
               00  0
               00  0
LD(BC),A       02  2
LD(DE),A       12  18

LD(HL),A       77  119
LD(HL),B       70  112
LD(HL),C       71  113
LD(HL),D       72  114
LD(HL),00      36  54
               00  0
LD(HL),E       73  115
LD(HL),H       74  116
LD(HL),L       75  117
LD(IX+00),A    DD  221
               77  119
               00  0
LD(IX+00),B    DD  221
               70  112
               00  0
LD(IX+00),C    DD  221
               71  113
               00  0
LD(IX+00),00   DD  221
               36  54
               00  0
               00  0
LD(IX+00),E    DD  221
               73  115
               00  0
LD(IX+00),H    DD  221
               74  116
               00  0
LD(IX+00),L    DD  221
               75  117
               00  0
LD(IY+00),A    FD  253
               77  119
               00  0
LD(IY+00),B    FD  253
               70  112
               00  0
LD(IY+00),C    FD  253
               71  113
               00  0
LD(IY+00),D    FD  253
               72  114
               00  0
LD(IY+00),00   FD  253
               36  54
               00  0
               00  0
LD(IY+00),E    FD  253
               73  115
               00  0
LD(IY+00),H    FD  253
               74  116
               00  0
LD(IY+00),L    FD  253
               75  117
               00  0
LD A,(0000)    3A  58
               00  0
               00  0
LD A,(BC)      0A  10
LD A,(DE)      1A  26
LD A,(HL)      7E  126
LD A,(IX+00)   DD  221
               7E  126
               00  0
LD A,(IY+00)   FD  253
               7E  126
               00  0
LD A,A         7F  127
LD A,B         78  120
LD A,C         79  121
LD A,D         7A  122
LD A,00        3E  62
               00  0
LD A,E         7B  123
LD A,H         7C  124
LD A,I         ED  237
               57  87
LD A,L         7D  125
LD A,R         ED  237
               5F  95
LD B,(HL)      46  70
```

```
LD B,(IX+00)    DD  221
                46  70
                00  0
LD B,(IY+00)    FD  253
                46  70
                00  0
LD B,A          47  71
LD B,B          40  64
LD B,C          41  65
LD B,D          42  66
LD B,00         06  6
                00  0
LD B,E          43  67
LD B,H          44  68
LD B,L          45  69
LD BC,(0000)    ED  237
                4B  75
                00  0
                00  0
LD BC,0000      01  1
                00  0
                00  0
LD C,(HL)       4E  78
LD C,(IX+00)    DD  221
                4E  78
                00  0
LD C,(IY+00)    FD  253
                4E  78
                00  0
LD C,A          4F  79
LD C,B          48  72
LD C,C          49  73
LD C,D          4A  74
LD C,00         0E  14
                00  0
LD C,E          4B  75
LD C,H          4C  76
LD C,L          4D  77
LD D,(HL)       56  86
LD D,(IX+00)    DD  221
                56  86
                00  0
LD D,(IY+00)    FD  253
                56  86
                00  0
LD D,A          57  87
LD D,B          50  80
LD D,C          51  81
LD D,D          52  82
LD D,00         16  22
                00  0
LD D,E          53  83
LD D,H          54  84
LD D,L          55  85
LD DE,(0000)    ED  237
                5B  91
                00  0
                00  0
LD DE,0000      11  17
                00  0
                00  0
LD E,(HL)       5E  94
LD E,(IX+00)    DD  221
                5E  94
                00  0
LD E,(IY+00     FD  253
                5E  94
                00  0
LD E,A          5F  95
LD E,B          58  88
LD E,C          59  89
LD E,D          5A  90
LD E,00         1E  30
                00  0
LD E,E          5B  91
LD E,H          5C  92
LD E,L          5D  93
LD H,(HL)       66  102
LD H,(IX+00)    DD  221
                66  102
                00  0
LD H,(IY+00)    FD  253
                66  102
                00  0
LD H,A          67  103
LD H,B          60  96
LD H,C          61  97
LD H,D          62  98
LD H,00         26  38
                00  0
LD H,E          63  99
LD H,H          64  100
LD H,L          65  101
LD HL,(0000)    ED  237
                6B  107
                00  0
                00  0
LD HL,(0000)    2A  42
                00  0
                00  0
LD HL,0000      21  33
                00  0
                00  0
LD I,A          ED  237
                47  71
LD IX,(0000)    DD  221
                2A  42
                00  0
                00  0
LD IX,0000      DD  221
                21  33
                00  0
                00  0
LD IY,(0000)    FD  253
                2A  42
                00  0
                00  0
LD IY,0000      FD  253
                21  33
                00  0
                00  0
LD L,(H_        6E  110
LD L,(IX+00)    DD  221
                6E  110
                00  0
LD L,(IY+00)    FD  253
                6E  110
                00  0
LD L,A          6F  111
LD L,B          68  104
LD L,C          69  105
LD L,D          6A  106
LD L,00         2E  46
                00  0
LD L,E          6B  107
LD L,H          6C  108
LD L,L          6D  109
LD R,A          ED  237
                4F  79
LD SP,(0000)    ED  237
                7B  123
                00  0
                00  0
LD SP,0000      31  49
                00  0
                00  0
LD SP,HL        F9  249
LD SP,IX        DD  221
                F9  249
LD SP,IY        FD  253
                F9  249
LDD             ED  237
                A8  168
LDDR            ED  237
                B8  184
LDI             ED  237
                A0  160
LDIR            ED  237
                B0  176
NEG             ED  237
                44  68
NOP             00  0
OR(HL)          B6  182
```

OR(IX+00)	DD	221
	B6	182
	00	0
OR(IY+00)	FD	253
	B6	182
	00	0
OR A	B7	183
OR B	B0	176
OR C	B1	177
OR D	B2	178
OR 00	F6	246
	00	0
OR E	B3	179
OR H	B4	180
OR L	B5	181
OTDR	ED	237
	BB	187
OTIR	ED	237
	B3	179
OUT(C),A	ED	237
	79	121
OUT(C),B	ED	237
	41	65
OUT(C),C	ED	237
	49	73
OUT(C),D	ED	237
	51	81
OUT(C),E	ED	237
	59	89
OUT(C),H	ED	237
	61	97
OUT(C),L	ED	237
	69	105
OUT(Port),A	D3	211
	00	0
OUTD	ED	237
	AB	171
OUTI	ED	237
	A3	163
POP AF	F1	241
POP BC	C1	193
POP DE	D1	209
POP HL	E1	225
POP IX	DD	221
	E1	225
POP IY	FD	253
	E1	225
PUSH AF	F5	245
PUSH BC	C5	197
PUSH DE	D5	213
PUSH HL	E5	229
PUSH IX	DD	221
	E5	229
PUSH IY	FD	253
	E5	229
RES 0,(HL)	CB	203
	86	134
RES 0,(IX+00)	DD	221
	CB	203
	00	0
	86	134
RES 0,(1Y+00)	FD	253
	CB	203
	00	0
	86	134
RES 0,A	CB	203
	87	135
RES 0,B	CB	203
	80	128
RES 0,C	CB	203
	81	129
RES 0,D	CB	203
	82	130
RES 0,E	CB	203
	83	131
RES 0,H	CB	203
	84	132
RES 0,L	CB	203
	85	133
RES 1,(HL)	CB	203
	8E	142

RES 1,(IX+00)	DD	221
	CB	203
	00	0
	8D	141
RES 1,(IY+00)	FD	253
	CB	203
	00	0
	8D	141
RES 1,A	CB	203
	8F	143
RES 1,B	CB	203
	88	136
RES 1,C	CB	203
	89	137
RES 1,D	CB	203
	8A	138
RES 1,E	CB	203
	8B	139
RES 1,H	CB	203
	8C	140
RES 1,L	CB	203
	8D	141
RES 2,(HL)	CB	203
	96	150
RES 2,(IX+00)	DD	221
	CB	203
	00	0
	96	150
RES 2,(IY+00)	FD	253
	CB	203
	00	0
	96	150
RES 2,A	CB	203
	97	151
RES 2,B	CB	203
	90	144
RES 2,C	CB	203
	91	145
RES 2,D	CB	203
	92	146
RES 2,E	CB	203
	93	147
RES 2,H	CB	203
	94	148
RES 5,C	CB	203
	A9	169
RES 5,D	CB	203
	AA	170
RES 5,E	CB	203
	AB	171
RES 5,H	CB	203
	AC	172
RES 5,L	CB	203
	AD	173
RES 6,(HL)	CB	203
	B6	182
RES 6,(IX+00)	DD	221
	CB	203
	00	0
	B6	182
RES 6,(IY+00)	FD	253
	CB	203
	00	0
	B6	182
RES 6,A	CB	203
	B7	183
RES 6,B	CB	203
	B0	176
RES 6,C	CB	203
	B1	177
RES 6,D	CB	203
	B2	178
RES 6,E	CB	203
	B3	179
RES 6,H	CB	203
	B4	180
RES 6,L	CB	203
	B5	181
RES 7,(HL)	CB	203
	BE	190

```
RES 7,(IX+00)  DD  221
               CB  203
               00  0
               BE  190
RES 7,(IY+00)  FD  253
               CB  203
               00  0
               BE  190
RES 7,A        CB  203
               BF  191
RES 7,B        CB  203
               B8  184
RES 7,C        CB  203
               B9  185
RES 7,D        CB  203
               BA  186
RES 7,E        CB  203
               BB  187
RES 7,H        CB  203
               BC  188
RES 7,L        CB  203
               BD  189
RET            C9  201
RET C          D8  216
RET M          F8  248
RET NC         D0  208
RET NZ         C0  192
RET P          F0  240
RES 2,L        CB  203
               95  149
RES 3,(HL)     CB  203
               9E  158
RES 3,(IX+00)  DD  221
               CB  203
               00  0
               9E  158
RES 3,(IY+00)  FD  253
               CB  203
               00  0
               9E  158
RES 3,A        CB  203
               9F  159
RES 3,B        CB  203
               98  152
RES 3,C        CB  203
               99  153
RES 3,D        CB  203
               9A  154
RES 3,E        CB  203
               9B  155
RES 3,H        CB  203
               9C  156
RES 3,L        CB  203
               9D  157
RES 4,(HL)     CB  203
               A6  166
RES 4,(IX+00)  DD  221
               CB  203
               00  0
               A6  166
RES 4,(IY+00)  FD  253
               CB  203
               00  0
               A6  166
RES 4,A        CB  203
               A7  167
RES 4,B        CB  203
               A0  160
RES 4,C        CB  203
               A1  161
RES 4,D        CB  203
               A2  162
RES 4,E        CB  203
               A3  163
RES 4,H        CB  203
               A4  164
RES 4,L        CB  203
               A5  165
RES 5,(HL)     CB  203
               AE  174
```

```
RES 5,(IX+00)  DD  221
               CB  203
               00  0
               AE  174
RES 5,(IY+00)  FD  253
               CB  203
               00  0
               AE  174
RES 5,A        CB  203
               5F  95
RES 5,B        CB  203
               A8  168
RET PE         E8  232
RET PO         E0  224
RET Z          C8  200
RETI           ED  237
               4D  77.
RETN           ED  237
               45  69
RL(HL)         CB  203
               16  22
RL(IX+00)      DD  221
               CB  203
               00  0
               16  22
RL(IY+00)      FD  253
               CB  203
               00  0
               16  22
RL A           CB  203
               17  23
RL B           CB  203
               10  16
RL C           CB  203
               11  17
RL D           CB  203
               12  18
RL E           CB  203
               13  19
RL H           CB  203
               14  20
RL L           CB  203
               15  21
RLA            17  23
RLC(HL)        CB  203
               06  6
RLC(IX+00)     DD  221
               CB  203
               00  0
               06  6
RLC(IY+00)     FD  253
               CB  203
               00  0
               06  6
RLC A          CB  203
               07  7
RLC B          CB  203
               00  0
RLC C          CB  203
               01  1
RLC D          CB  203
               02  2
RLC E          CB  203
               03  3
RLC H          CB  203
               04  4
RLC L          CB  203
               05  5
RLCA           07  7
RLD            ED  237
               6F  111
RR(HL)         CB  203
               1E  30
RR(IX+00)      DD  221
               CB  203
               00  0
               1E  30
RR(IY+00)      FD  253
               CB  203
               00  0
               1E  30
```

```
RR A            CB  203
                1F  31
RR B            CB  203
                18  24
RR C            CB  203
                19  25
RR D            CB  203
                1A  26
RR E            CB  203
                1B  27
RR H            CB  203
                1C  28
RR L            CB  203
                1D  29
RRA             1F  31
RRC(HL)         CB  203
                0E  14
RRC(IX+00)      DD  221
                CB  203
                00  0
                0E  14
RRC(IY+00)      FD  253
                CB  203
                00  0
                0E  14
RRC A           CB  203
                0F  15
RRC B           CB  203
                08  8
RRC C           CB  203
                09  9
RRC D           CB  203
                0A  10
RRC E           CB  203
                0B  11
RRC H           CB  203
                0C  12
RRC L           CB  203
                0D  13
RRCA            0F  15
RRD             ED  237
                67  103
RST 00          C7  199
RST 08          CF  207
RST 10          D7  215
RST 18          DF  223
RST 20          E7  231
RST 28          EF  239
RST 30          F7  247
RST 38          FF  255
SBC A,(HL)      9E  158
SBC A,(IX+00)   DD  221
                9E  158
                00  0
SBC A,(IY+00)   FD  253
                9E  158
                00  0
SBC A,A         9F  159
SBC A,B         98  152
SBC A,C         99  153
SBC A,D         9A  154
SBC A,00        DE  222
                00  0
SBC A,E         9B  155
SBC A,H         9C  156
SBC A,L         9D  157
SBC HL,BC       ED  237
                42  66
SBC HL,DE       ED  237
                52  82
SBC HL,HL       ED  237
                62  98
SBC HL,SP       ED  237
                72  114
SCF             37  55
SET 0,(HL)      CB  203
                C6  198
SET0,(IX+00)    DD  221
                CB  203
                00  0
                C6  198
SET0,(IY+00)    FD  253
                CB  203
                00  0
                C6  198
SET0,A          CB  203
                C7  199
SET0,B          CB  203
                C0  192
SET0,C          CB  203
                C1  193
SET0,D          CB  203
                C2  194
SET0,E          CB  203
                C3  195
SET0,H          CB  203
                C4  196
SET0,L          CB  203
                C5  197
SET1,(HL)       CB  203
                CE  206
SET1,(IX+00)    DD  221
                CB  203
                00  0
                CE  206
SET1,(IY+00)    FD  253
                CB  203
                00  0
                CE  206
SET1,A          CB  203
                CF  207
SET1,B          CB  203
                C8  200
SET1,C          CB  203
                C9  201
SET1,D          CB  203
                CA  202
SET1,E          CB  203
                CB  203
SET1,H          CB  203
                CC  204
SET1,L          CB  203
                CD  205
SET2,(HL)       CB  203
                D6  214
SET2,(IX+00)    DD  221
                CB  203
                00  0
                D6  214
SET2,(IY+00)    FD  253
                CB  203
                00  0
                D6  214
SET2,A          CB  203
                D7  215
SET2,B          CB  203
                D0  208
SET2,C          CB  203
                D1  209
SET2,D          CB  203
                D2  210
SET2,E          CB  203
                D3  211
SET2,H          CB  203
                D4  212
SET2,L          CB  203
                D5  213
SET3,(HL)       CB  203
                DE  222
SET3,(IX+00)    DD  221
                CB  203
                00  0
                DE  222
SET3,(IY+00)    FD  253
                CB  203
                00  0
                DE  222
SET3,A          CB  203
                DF  223
SET3,B          CB  203
                D8  216
SET3,C          CB  203
                D9  217
```

SET3,D	CB	203
	DA	218
SET3,E	CB	203
	DB	219
SET3,H	CB	203
	DC	220
SET3,L	CB	203
	DD	221
SET4,(HL)	CB	203
	E6	230
SET4,(IX+00)	DD	221
	CB	203
	00	0
	E6	230
SET4,(IY+00)	FD	253
	CB	203
	00	0
	E6	230
SET4,A	CB	203
	E7	231
SET4,B	CB	203
	E0	224
SET4,C	CB	203
	E1	225
SET4,D	CB	203
	E2	226
SET4,E	CB	203
	E3	227
SET4,H	CB	203
	E4	228
SET4,L	CB	203
	E5	229
SET5,(HL)	CB	203
	EE	238
SET5,(IX+00)	DD	221
	CB	203
	00	0
	EE	238
SET5,(IY+00)	FD	253
	CB	203
	00	0
	EE	238
SET5,A	CB	203
	EF	239
SET5,B	CB	203
	E8	232
SET5,C	CB	203
	E9	233
SET5,D	CB	203
	EA	234
SET5,E	CB	203
	EB	235
SET5,H	CB	203
	EC	236
SET5,L	CB	203
	ED	237
SET6,(HL)	CB	203
	F6	246
SET6,(IX+00)	DD	221
	CB	203
	00	0
	F6	246
SET6,(IY+00)	FD	253
	CB	203
	00	0
	F6	246
SET6,A	CB	203
	F7	247
SET6,B	CB	203
	F0	240
SET6,C	CB	203
	F1	241
SET6,D	CB	203
	F2	242
SET6,E	CB	203
	F3	243
SET6,H	CB	203
	F4	244
SET6,L	CB	203
	F5	245
SET7,(HL)	CB	203
	FE	254
SET7,(IX+00)	DD	221
	CB	203
	00	0
	FE	254
SET7,(IY+00)	FD	253
	CB	203
	00	0
	FE	254
SET7,A	CB	203
	FF	255
SET7,B	CB	203
	F8	248
SET7,C	CB	203
	F9	249
SET7,D	CB	203
	FA	250
SET7,E	CB	203
	FB	251
SET7,H	CB	203
	FC	252
SET7,L	CB	203
	FD	253
SLA(HL)	CB	203
	26	38
SLA(IX+00)	DD	221
	CB	203
	00	0
	26	38
SLA(IY+00)	FD	253
	CB	203
	00	0
	26	38
SLA A	CB	203
	27	39
SLA B	CB	203
	20	32
SLA C	CB	203
	21	33
SLA D	CB	203
	22	34
SLA E	CB	203
	23	35
SLA H	CB	203
	24	36
SLA L	CB	203
	25	37
SRA(HL)	CB	203
	2E	46
SRA(IX+00)	DD	221
	CB	203
	00	0
	2E	46
SRA(IY+00)	FD	253
	CB	203
	00	0
	2E	46
SRA A	CB	203
	2F	47
SRA B	CB	203
	28	40
SRA C	CB	203
	29	41
SRA D	CB	203
	2A	42
SRA E	CB	203
	2B	43
SRA H	CB	203
	2C	44
SRA L	CB	203
	2D	45
SRL(HL)	CB	203
	3E	62
SRL(IX+00)	DD	221
	CB	203
	00	0
	3E	62
SRL(IY+00)	FD	253
	CB	203
	00	0
	3E	62
SRL A	CB	203
	3F	63

SRL B	CB	203
	38	56
SRL C	CB	203
	39	57
SRL D	CB	203
	3A	58
SRL E	CB	203
	3B	59
SRL H	CB	203
	3C	60
SRL L	CB	203
	3D	61
SUB(HL)	96	150
SUB(IX+00)	DD	221
	96	150
	00	0
SUB(IY+00)	FD	253
	96	150
	00	0
SUB A	97	151
SUB B	90	144
SUB C	91	145

SUB D	92	146
SUB 00	D6	214
	00	0
SUB E	93	147
SUB H	94	148
SUB L	95	149
XOR(HL)	AE	174
XOR(IX+00)	DD	221
	AE	174
	00	0
XOR(IY+00)	FD	253
	AE	174
	00	0
XOR A	AF	175
XOR B	A8	168
XOR C	A9	169
XOR D	AA	170
XOR 00	EE	238
	00	0
XOR E	AB	171
XOR H	AC	172
XOR L	AD	173

Index

Introducing Z-80 Assembly Language Programming

This book has been written for newcomers to assembly language programming. The reader is likely to have some experience in programming in BASIC, but will have come to realise its shortcomings, many of which can be overcome by use of assembly language.

Practical methods of designing and entering assembly code are emphasised, rather than the detailed study of each command, and the interaction between machine code and hardware is stressed.

The book will be of interest to all users of Z-80 based micros, including the ZX81, ZX Spectrum, TRS80, Video Genie etc.

ISBN 0 408 01338 9

Newnes Microcomputer Books

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/